目錄

TutorialsPoint NumPy 教程

> 来源：NumPy Tutorial - TutorialsPoint
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

- 在线阅读
- PDF格式
- EPUB格式
- MOBI格式
- 代码仓库

赞助我

NumPy - 简介

NumPy 是一个 Python 包。 它代表 “Numeric Python”。 它是一个由多维数组对象和用于处理数组的例程集合组成的库。

Numeric，即 NumPy 的前身，是由 Jim Hugunin 开发的。 也开发了另一个包 Numarray ，它拥有一些额外的功能。 2005年，Travis Oliphant 通过将 Numarray 的功能集成到 Numeric 包中来创建 NumPy 包。 这个开源项目有很多贡献者。

NumPy 操作

使用NumPy，开发人员可以执行以下操作：

- 数组的算数和逻辑运算。
- 傅立叶变换和用于图形操作的例程。
- 与线性代数有关的操作。 NumPy 拥有线性代数和随机数生成的内置函数。

NumPy – MatLab 的替代之一

NumPy 通常与 **SciPy**（Scientific Python）和 **Matplotlib**（绘图库）一起使用。 这种组合广泛用于替代 MatLab，是一个流行的技术计算平台。 但是，Python 作为 MatLab 的替代方案，现在被视为一种更加现代和完整的编程语言。

NumPy 是开源的，这是它的一个额外的优势。

NumPy - 环境

> ### 在线尝试
>
> 我们已经在线设置了 NumPy 编程环境，以便在线编译和执行所有可用的示例。 它向你提供了信心，并使您能够使用不同的选项验证程序， 随意修改任何示例并在线执行。
>
> 使用我们的在线编译器尝试一下示例，它位于 CodingGround
>
> ```
> import numpy as np
> a =  'hello world'
> print a
> ```
>
> 对于本教程中给出的大多数示例，你会在我们的网站代码部分的右上角找到一个 `Try it` 选项，这会把你带到在线编译器。 所以快来使用它，享受你的学习吧。

标准的 Python 发行版不会与 NumPy 模块捆绑在一起。 一个轻量级的替代方法是使用流行的 Python 包安装程序 **pip** 来安装 NumPy。

```
pip install numpy
```

启用 NumPy 的最佳方法是使用特定于您的操作系统的可安装的二进制包。 这些二进制包含完整的 SciPy 技术栈（包括 NumPy，SciPy，matplotlib，IPython，SymPy 以及 Python 核心自带的其它包）。

Windows

Anaconda (from www.continuum.io) 是一个带有 SciPy 技术栈的免费 Python 发行版。 它也可用于 Linux 和 Mac.

Canopy (www.enthought.com/products/canopy/) 是可用的免费和商业发行版，带有完整的 SciPy 技术栈，可用于 Windows, Linux and Mac。

Python (x,y): 是个免费的 Python 发行版，带有 SciPy 技术栈和 Spyder IDE，可用于 Windows。 (从这里下载：www.python-xy.github.io/)

Linux

Linux 发行版的相应软件包管理器可用于安装一个或多个 SciPy 技术栈中的软件包。

对于 Ubuntu

```
sudo apt-get install python-numpy
python-scipy python-matplotlibipythonipythonnotebook python-pand
as
python-sympy python-nose
```

对于 Fedora

```
sudo yum install numpyscipy python-matplotlibipython
python-pandas sympy python-nose atlas-devel
```

从源码构建

核心 Python（2.6.x，2.7.x 和 3.2.x 起）必须安装`distutils`，`zlib`模块应该启用。

GNU gcc（4.2及以上）C 编译器必须可用。

要安装 NumPy，请运行以下命令。

```
Python setup.py install
```

要测试 NumPy 模块是否正确安装，请尝试从 Python 提示符导入它。

如果未安装，将显示以下错误消息。

```
Traceback (most recent call last):
   File "<pyshell#0>", line 1, in <module>
      import numpy
ImportError: No module named 'numpy'
```

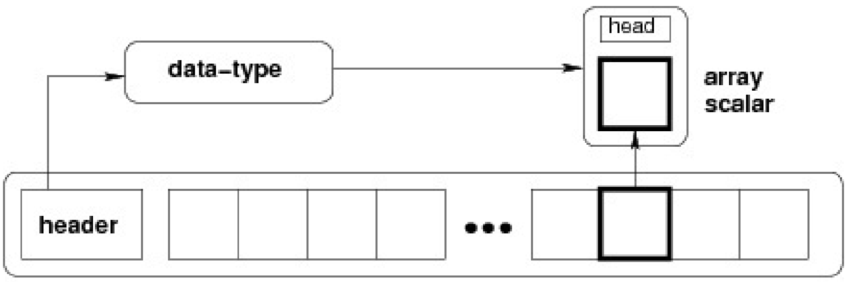
或者，使用以下语法导入NumPy包。

NumPy - Ndarray 对象

NumPy 中定义的最重要的对象是称为 `ndarray` 的 N 维数组类型。 它描述相同类型的元素集合。 可以使用基于零的索引访问集合中的项目。

`ndarray` 中的每个元素在内存中使用相同大小的块。 `ndarray` 中的每个元素是数据类型对象的对象（称为 `dtype` ）。

从 `ndarray` 对象提取的任何元素（通过切片）由一个数组标量类型的 Python 对象表示。 下图显示了 `ndarray` ，数据类型对象（ `dtype` ）和数组标量类型之间的关系。



`ndarray` 类的实例可以通过本教程后面描述的不同的数组创建例程来构造。 基本的 `ndarray` 是使用 NumPy 中的数组函数创建的，如下所示：

```
numpy.array
```

它从任何暴露数组接口的对象，或从返回数组的任何方法创建一个ndarray。

```
numpy.array(object, dtype = None, copy = True, order = None, sub
ok = False, ndmin = 0)
```

上面的构造器接受以下参数：

序号	参数及描述
1.	`object` 任何暴露数组接口方法的对象都会返回一个数组或任何（嵌套）序列。
2.	`dtype` 数组的所需数据类型，可选。
3.	`copy` 可选，默认为 `true` ，对象是否被复制。
4.	`order` C （按行）、 F （按列）或 A （任意，默认）。
5.	`subok` 默认情况下，返回的数组被强制为基类数组。 如果为 `true` ，则返回子类。
6.	`ndimin` 指定返回数组的最小维数。

看看下面的例子来更好地理解。

示例 1

```
import numpy as np
a = np.array([1,2,3])
print a
```

输出如下：

```
[1, 2, 3]
```

示例 2

```
# 多于一个维度
import numpy as np
a = np.array([[1,  2],  [3,  4]])
print a
```

输出如下：

```
[[1, 2]
 [3, 4]]
```

示例 3

```
# 最小维度
import numpy as np
a = np.array([1,  2,  3,4,5], ndmin =  2)
print a
```

输出如下：

```
[[1, 2, 3, 4, 5]]
```

示例 4

```
# dtype 参数
import numpy as np
a = np.array([1,  2,  3], dtype = complex)
print a
```

输出如下：

```
[ 1.+0.j,  2.+0.j,  3.+0.j]
```

ndarray 对象由计算机内存中的一维连续区域组成，带有将每个元素映射到内存块中某个位置的索引方案。 内存块以按行（C 风格）或按列（FORTRAN 或 MatLab 风格）的方式保存元素。

NumPy - 数据类型

NumPy 支持比 Python 更多种类的数值类型。 下表显示了 NumPy 中定义的不同标量数据类型。

序号	数据类型及描述
1.	`bool_` 存储为一个字节的布尔值（真或假）
2.	`int_` 默认整数，相当于 C 的 `long` ，通常为 `int32` 或 `int64`
3.	`intc` 相当于 C 的 `int` ，通常为 `int32` 或 `int64`
4.	`intp` 用于索引的整数，相当于 C 的 `size_t` ，通常为 `int32` 或 `int64`
5.	`int8` 字节（-128 ~ 127）
6.	`int16` 16 位整数（-32768 ~ 32767）
7.	`int32` 32 位整数（-2147483648 ~ 2147483647）
8.	`int64` 64 位整数（-9223372036854775808 ~ 9223372036854775807）
9.	`uint8` 8 位无符号整数（0 ~ 255）
10.	`uint16` 16 位无符号整数（0 ~ 65535）
11.	`uint32` 32 位无符号整数（0 ~ 4294967295）
12.	`uint64` 64 位无符号整数（0 ~ 18446744073709551615）
13.	`float_` `float64` 的简写
14.	`float16` 半精度浮点：符号位，5 位指数，10 位尾数
15.	`float32` 单精度浮点：符号位，8 位指数，23 位尾数
16.	`float64` 双精度浮点：符号位，11 位指数，52 位尾数
17.	`complex_` `complex128` 的简写
18.	`complex64` 复数，由两个 32 位浮点表示（实部和虚部）
19.	`complex128` 复数，由两个 64 位浮点表示（实部和虚部）

NumPy 数字类型是 `dtype` （数据类型）对象的实例，每个对象具有唯一的特征。这些类型可以是 `np.bool_` ， `np.float32` 等。

数据类型对象 (`dtype`)

数据类型对象描述了对应于数组的固定内存块的解释，取决于以下方面：

- 数据类型（整数、浮点或者 Python 对象）
- 数据大小
- 字节序（小端或大端）
- 在结构化类型的情况下，字段的名称，每个字段的数据类型，和每个字段占用的内存块部分。
- 如果数据类型是子序列，它的形状和数据类型。

字节顺序取决于数据类型的前缀 `<` 或 `>` 。 `<` 意味着编码是小端（最小有效字节存储在最小地址中）。 `>` 意味着编码是大端（最大有效字节存储在最小地址中）。

`dtype` 可由一下语法构造：

```
numpy.dtype(object, align, copy)
```

参数为：

- `Object` ：被转换为数据类型的对象。
- `Align` ：如果为 `true` ，则向字段添加间隔，使其类似 C 的结构体。
- `Copy` ？生成 `dtype` 对象的新副本，如果为 `flase` ，结果是内建数据类型对象的引用。

示例 1

```
# 使用数组标量类型
import numpy as np
dt = np.dtype(np.int32)
print dt
```

输出如下：

```
int32
```

示例 2

```
#int8，int16，int32，int64 可替换为等价的字符串 'i1'，'i2'，'i4'，以及
其他。
import numpy as np

dt = np.dtype('i4')
print dt
```

输出如下：

```
int32
```

示例 3

```
# 使用端记号
import numpy as np
dt = np.dtype('>i4')
print dt
```

输出如下：

```
>i4
```

下面的例子展示了结构化数据类型的使用。 这里声明了字段名称和相应的标量数据类型。

示例 4

```
# 首先创建结构化数据类型。
import numpy as np
dt = np.dtype([('age',np.int8)])
print dt
```

输出如下：

```
[('age', 'i1')]
```

示例 5

```
# 现在将其应用于 ndarray 对象
import numpy as np

dt = np.dtype([('age',np.int8)])
a = np.array([(10,),(20,),(30,)], dtype = dt)
print a
```

输出如下：

```
[(10,) (20,) (30,)]
```

示例 6

```
# 文件名称可用于访问 age 列的内容
import numpy as np

dt = np.dtype([('age',np.int8)])
a = np.array([(10,),(20,),(30,)], dtype = dt)
print a['age']
```

输出如下：

```
[10 20 30]
```

示例 7

以下示例定义名为 **student** 的结构化数据类型，其中包含字符串字段 `name` ，整数字段 `age` 和浮点字段 `marks` 。此 `dtype` 应用于 `ndarray` 对象。

```
import numpy as np
student = np.dtype([('name','S20'),  ('age',  'i1'),  ('marks',
 'f4')])
print student
```

输出如下：

```
[('name', 'S20'), ('age', 'i1'), ('marks', '<f4')])
```

示例 8

```
import numpy as np

student = np.dtype([('name','S20'),  ('age',  'i1'),  ('marks',
 'f4')])
a = np.array([('abc',  21,  50),('xyz',  18,  75)], dtype = stud
ent)
print a
```

输出如下：

```
[('abc', 21, 50.0), ('xyz', 18, 75.0)]
```

每个内建类型都有一个唯一定义它的字符代码：

- `'b'`：布尔值
- `'i'`：符号整数
- `'u'`：无符号整数
- `'f'`：浮点
- `'c'`：复数浮点
- `'m'`：时间间隔
- `'M'`：日期时间
- `'O'`：Python 对象
- `'S', 'a'`：字节串
- `'U'`：Unicode
- `'V'`：原始数据（`void`）

NumPy - 数组属性

这一章中，我们会讨论 NumPy 的多种数组属性。

`ndarray.shape`

这一数组属性返回一个包含数组维度的元组，它也可以用于调整数组大小。

示例 1

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
print a.shape
```

输出如下：

```
(2, 3)
```

示例 2

```
# 这会调整数组大小
import numpy as np

a = np.array([[1,2,3],[4,5,6]]) a.shape =  (3,2)
print a
```

输出如下：

```
[[1, 2]
 [3, 4]
 [5, 6]]
```

示例 3

NumPy 也提供了 `reshape` 函数来调整数组大小。

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
b = a.reshape(3,2)
print b
```

输出如下：

```
[[1, 2]
 [3, 4]
 [5, 6]]
```

`ndarray.ndim`

这一数组属性返回数组的维数。

示例 1

```
# 等间隔数字的数组
import numpy as np
a = np.arange(24)  print a
```

输出如下：

```
[0 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 17 18 1
9 20 21 22 23]
```

示例 2

```
# 一维数组
import numpy as np
a = np.arange(24) a.ndim
# 现在调整其大小
b = a.reshape(2,4,3)
print b
# b 现在拥有三个维度
```

输出如下：

```
[[[ 0,  1,  2]
  [ 3,  4,  5]
  [ 6,  7,  8]
  [ 9, 10, 11]]
  [[12, 13, 14]
   [15, 16, 17]
   [18, 19, 20]
   [21, 22, 23]]]
```

`numpy.itemsize`

这一数组属性返回数组中每个元素的字节单位长度。

示例 1

```
# 数组的 dtype 为 int8（一个字节）
import numpy as np
x = np.array([1,2,3,4,5], dtype = np.int8)
print x.itemsize
```

输出如下：

```
1
```

示例 2

```
# 数组的 dtype 现在为 float32（四个字节）
import numpy as np
x = np.array([1,2,3,4,5], dtype = np.float32)
print x.itemsize
```

输出如下：

```
4
```

`numpy.flags`

`ndarray` 对象拥有以下属性。这个函数返回了它们的当前值。

序号	属性及描述
1.	`C_CONTIGUOUS (C)` 数组位于单一的、C 风格的连续区段内
2.	`F_CONTIGUOUS (F)` 数组位于单一的、Fortran 风格的连续区段内
3.	`OWNDATA (O)` 数组的内存从其它对象处借用
4.	`WRITEABLE (W)` 数据区域可写入。将它设置为 `flase` 会锁定数据，使其只读
5.	`ALIGNED (A)` 数据和任何元素会为硬件适当对齐
6.	`UPDATEIFCOPY (U)` 这个数组是另一数组的副本。当这个数组释放时，源数组会由这个数组中的元素更新

示例

下面的例子展示当前的标志。

```
import numpy as np
x = np.array([1,2,3,4,5])
print x.flags
```

输出如下：

```
C_CONTIGUOUS : True
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

NumPy - 数组创建例程

新的 `ndarray` 对象可以通过任何下列数组创建例程或使用低级 `ndarray` 构造函数构造。

`numpy.empty`

它创建指定形状和 `dtype` 的未初始化数组。 它使用以下构造函数：

```
numpy.empty(shape, dtype = float, order = 'C')
```

构造器接受下列参数：

序号	参数及描述
1.	`Shape` 空数组的形状，整数或整数元组
2.	`Dtype` 所需的输出数组类型，可选
3.	`Order` `'C'` 为按行的 C 风格数组， `'F'` 为按列的 Fortran 风格数组

示例

下面的代码展示空数组的例子：

```
import numpy as np
x = np.empty([3,2], dtype =  int)
print x
```

输出如下：

```
[[22649312    1701344351]
 [1818321759  1885959276]
 [16779776    156368896]]
```

注意：数组元素为随机值，因为它们未初始化。

`numpy.zeros`

返回特定大小，以 0 填充的新数组。

```
numpy.zeros(shape, dtype = float, order = 'C')
```

构造器接受下列参数：

序号	参数及描述
1.	`Shape` 空数组的形状，整数或整数元组
2.	`Dtype` 所需的输出数组类型，可选
3.	`Order` `'C'` 为按行的 C 风格数组，`'F'` 为按列的 Fortran 风格数组

示例 1

```
# 含有 5 个 0 的数组，默认类型为 float
import numpy as np
x = np.zeros(5)
print x
```

输出如下：

```
[ 0.  0.  0.  0.  0.]
```

示例 2

```
import numpy as np
x = np.zeros((5,), dtype = np.int)
print x
```

输出如下：

```
[0  0  0  0  0]
```

示例 3

```
# 自定义类型
import numpy as np
x = np.zeros((2,2), dtype =  [('x',  'i4'),  ('y',  'i4')])
print x
```

输出如下：

```
[[(0,0)(0,0)]
 [(0,0)(0,0)]]
```

`numpy.ones`

返回特定大小，以 1 填充的新数组。

```
numpy.ones(shape, dtype = None, order = 'C')
```

构造器接受下列参数：

序号	参数及描述
1.	`Shape` 空数组的形状，整数或整数元组
2.	`Dtype` 所需的输出数组类型，可选
3.	`Order` `'C'` 为按行的 C 风格数组，`'F'` 为按列的 Fortran 风格数组

示例 1

```
# 含有 5 个 1 的数组，默认类型为 float
import numpy as np
x = np.ones(5)  print x
```

输出如下：

```
[ 1.  1.  1.  1.  1.]
```

示例 2

```
import numpy as np
x = np.ones([2,2], dtype =  int)
print x
```

输出如下：

```
[[1  1]
 [1  1]]
```

NumPy - 来自现有数据的数组

这一章中，我们会讨论如何从现有数据创建数组。

`numpy.asarray`

此函数类似于 `numpy.array` ，除了它有较少的参数。 这个例程对于将 Python 序列转换为 `ndarray` 非常有用。

```
numpy.asarray(a, dtype = None, order = None)
```

构造器接受下列参数：

序号	参数及描述
1.	`a` 任意形式的输入参数，比如列表、列表的元组、元组、元组的元组、元组的列表
2.	`dtype` 通常，输入数据的类型会应用到返回的 `ndarray`
3.	`order` `'C'` 为按行的 C 风格数组， `'F'` 为按列的 Fortran 风格数组

下面的例子展示了如何使用 `asarray` 函数：

示例 1

```
# 将列表转换为 ndarray
import numpy as np

x =  [1,2,3]
a = np.asarray(x)
print a
```

输出如下：

```
[1  2  3]
```

示例 2

```
# 设置了 dtype
import numpy as np

x =  [1,2,3]
a = np.asarray(x, dtype =  float)
print a
```

输出如下：

```
[ 1.  2.  3.]
```

示例 3

```
# 来自元组的 ndarray
import numpy as np

x =  (1,2,3)
a = np.asarray(x)
print a
```

输出如下：

```
[1  2  3]
```

示例 4

```
# 来自元组列表的 ndarray
import numpy as np

x =  [(1,2,3),(4,5)]
a = np.asarray(x)
print a
```

输出如下：

```
[(1, 2, 3) (4, 5)]
```

`numpy.frombuffer`

此函数将缓冲区解释为一维数组。 暴露缓冲区接口的任何对象都用作参数来返回 `ndarray` 。

```
numpy.frombuffer(buffer, dtype = float, count = -1, offset = 0)
```

构造器接受下列参数：

序号	参数及描述
1.	`buffer` 任何暴露缓冲区借口的对象
2.	`dtype` 返回数组的数据类型，默认为 `float`
3.	`count` 需要读取的数据数量，默认为 `-1` ，读取所有数据
4.	`offset` 需要读取的起始位置，默认为 `0`

示例

下面的例子展示了 `frombuffer` 函数的用法。

```
import numpy as np
s =  'Hello World'
a = np.frombuffer(s, dtype =  'S1')
print a
```

输出如下：

```
['H'  'e'  'l'  'l'  'o'  ' '  'W'  'o'  'r'  'l'  'd']
```

`numpy.fromiter`

此函数从任何可迭代对象构建一个 `ndarray` 对象，返回一个新的一维数组。

```
numpy.fromiter(iterable, dtype, count = -1)
```

构造器接受下列参数：

序号	参数及描述
1.	`iterable` 任何可迭代对象
2.	`dtype` 返回数组的数据类型
3.	`count` 需要读取的数据数量，默认为`-1`，读取所有数据

以下示例展示了如何使用内置的`range()`函数返回列表对象。 此列表的迭代器用于形成`ndarray`对象。

示例 1

```
# 使用 range 函数创建列表对象
import numpy as np
list = range(5)
print list
```

输出如下：

```
[0,  1,  2,  3,  4]
```

示例 2

```
# 从列表中获得迭代器
import numpy as np
list = range(5)
it = iter(list)
# 使用迭代器创建 ndarray
x = np.fromiter(it, dtype =  float)
print x
```

输出如下：

```
[0.   1.   2.   3.   4.]
```

NumPy - 来自数值范围的数组

这一章中，我们会学到如何从数值范围创建数组。

`numpy.arange`

这个函数返回`ndarray`对象，包含给定范围内的等间隔值。

```
numpy.arange(start, stop, step, dtype)
```

构造器接受下列参数：

序号	参数及描述
1.	`start` 范围的起始值，默认为`0`
2.	`stop` 范围的终止值（不包含）
3.	`step` 两个值的间隔，默认为`1`
4.	`dtype` 返回 `ndarray` 的数据类型，如果没有提供，则会使用输入数据的类型。

下面的例子展示了如何使用该函数：

示例 1

```
import numpy as np
x = np.arange(5)
print x
```

输出如下：

```
[0  1  2  3  4]
```

示例 2

```
import numpy as np
# 设置了 dtype
x = np.arange(5, dtype =  float)
print x
```

输出如下：

```
[0.  1.  2.  3.  4.]
```

示例 3

```
# 设置了起始值和终止值参数
import numpy as np
x = np.arange(10,20,2)
print x
```

输出如下：

```
[10  12  14  16  18]
```

`numpy.linspace`

此函数类似于 `arange()` 函数。 在此函数中，指定了范围之间的均匀间隔数量，而不是步长。 此函数的用法如下。

```
numpy.linspace(start, stop, num, endpoint, retstep, dtype)
```

构造器接受下列参数：

序号	参数及描述
1.	`start` 序列的起始值
2.	`stop` 序列的终止值，如果 `endpoint` 为 `true` ，该值包含于序列中
3.	`num` 要生成的等间隔样例数量，默认为 `50`
4.	`endpoint` 序列中是否包含 `stop` 值，默认为 `ture`
5.	`retstep` 如果为 `true` ，返回样例，以及连续数字之间的步长
6.	`dtype` 输出 `ndarray` 的数据类型

下面的例子展示了 `linspace` 函数的用法。

示例 1

```
import numpy as np
x = np.linspace(10,20,5)
print x
```

输出如下：

```
[10.   12.5   15.   17.5  20.]
```

示例 2

```
# 将 endpoint 设为 false
import numpy as np
x = np.linspace(10,20,  5, endpoint =  False)
print x
```

输出如下：

```
[10.   12.   14.   16.   18.]
```

示例 3

```
# 输出 retstep 值
import numpy as np

x = np.linspace(1,2,5, retstep =  True)
print x
# 这里的 retstep 为 0.25
```

输出如下：

```
(array([ 1.  ,  1.25,  1.5 ,  1.75,  2.  ]), 0.25)
```

`numpy.logspace`

此函数返回一个`ndarray`对象，其中包含在对数刻度上均匀分布的数字。 刻度的开始和结束端点是某个底数的幂，通常为 10。

```
numpy.logspace(start, stop, num, endpoint, base, dtype)
```

`logspace`函数的输出由以下参数决定：

序号	参数及描述
1.	`start` 起始值是`base ** start`
2.	`stop` 终止值是 `base ** stop`
3.	`num` 范围内的数值数量，默认为`50`
4.	`endpoint` 如果为 `true` ，终止值包含在输出数组当中
5.	`base` 对数空间的底数，默认为`10`
6.	`dtype` 输出数组的数据类型，如果没有提供，则取决于其它参数

下面的例子展示了`logspace`函数的用法。

示例 1

```
import numpy as np
# 默认底数是 10
a = np.logspace(1.0,  2.0, num =  10)
print a
```

输出如下：

```
[ 10.           12.91549665     16.68100537      21.5443469  27.
82559402
  35.93813664   46.41588834     59.94842503      77.42636827
100.    ]
```

示例 2

```
# 将对数空间的底数设置为 2
import numpy as np
a = np.logspace(1,10,num =  10,  base  =  2)
print a
```

输出如下：

```
[ 2.      4.      8.     16.     32.     64.    128.    256.     512.
1024.]
```

NumPy - 切片和索引

`ndarray` 对象的内容可以通过索引或切片来访问和修改，就像 Python 的内置容器对象一样。

如前所述，`ndarray` 对象中的元素遵循基于零的索引。 有三种可用的索引方法类型： 字段访问，基本切片和高级索引。

基本切片是 Python 中基本切片概念到 n 维的扩展。 通过将 `start`，`stop` 和 `step` 参数提供给内置的 `slice` 函数来构造一个 Python `slice` 对象。 此 `slice` 对象被传递给数组来提取数组的一部分。

示例 1

```
import numpy as np
a = np.arange(10)
s = slice(2,7,2)
print a[s]
```

输出如下：

```
[2  4  6]
```

在上面的例子中，`ndarray` 对象由 `arange()` 函数创建。 然后，分别用起始，终止和步长值 `2`，`7` 和 `2` 定义切片对象。 当这个切片对象传递给 `ndarray` 时，会对它的一部分进行切片，从索引 `2` 到 `7`，步长为 `2`。

通过将由冒号分隔的切片参数（`start:stop:step`）直接提供给 `ndarray` 对象，也可以获得相同的结果。

示例 2

```
import numpy as np
a = np.arange(10)
b = a[2:7:2]
print b
```

输出如下：

```
[2  4  6]
```

如果只输入一个参数，则将返回与索引对应的单个项目。 如果使用 `start:` ，则从该索引向后的所有项目将被提取。 如果使用两个参数（以 `:` 分隔），则对两个索引（不包括停止索引）之间的元素以默认步骤进行切片。

示例 3

```
# 对单个元素进行切片
import numpy as np

a = np.arange(10)
b = a[5]
print b
```

输出如下：

```
5
```

示例 4

```
# 对始于索引的元素进行切片
import numpy as np
a = np.arange(10)
print a[2:]
```

输出如下：

```
[2  3  4  5  6  7  8  9]
```

示例 5

```
# 对索引之间的元素进行切片
import numpy as np
a = np.arange(10)
print a[2:5]
```

输出如下：

```
[2  3  4]
```

上面的描述也可用于多维 `ndarray` 。

示例 6

```
import numpy as np
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
print a
# 对始于索引的元素进行切片
print  '现在我们从索引 a[1:] 开始对数组切片'
print a[1:]
```

输出如下：

```
[[1 2 3]
 [3 4 5]
 [4 5 6]]

现在我们从索引 a[1:] 开始对数组切片
[[3 4 5]
 [4 5 6]]
```

切片还可以包括省略号（ `...` ），来使选择元组的长度与数组的维度相同。 如果在行位置使用省略号，它将返回包含行中元素的 `ndarray` 。

示例 7

```
# 最开始的数组
import numpy as np
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
print  '我们的数组是：'
print a
print  '\n'
# 这会返回第二列元素的数组：
print  '第二列的元素是：'
print a[...,1]
print  '\n'
# 现在我们从第二行切片所有元素：
print  '第二行的元素是：'
print a[1,...]
print  '\n'
# 现在我们从第二列向后切片所有元素：
print  '第二列及其剩余元素是：'
print a[...,1:]
```

输出如下：

```
我们的数组是：
[[1 2 3]
 [3 4 5]
 [4 5 6]]

第二列的元素是：
[2 4 5]

第二行的元素是：
[3 4 5]

第二列及其剩余元素是：
[[2 3]
 [4 5]
 [5 6]]
```

NumPy - 高级索引

如果一个 `ndarray` 是非元组序列，数据类型为整数或布尔值的 `ndarray` ，或者至少一个元素为序列对象的元组，我们就能够用它来索引 `ndarray` 。高级索引始终返回数据的副本。 与此相反，切片只提供了一个视图。

有两种类型的高级索引：整数和布尔值。

整数索引

这种机制有助于基于 N 维索引来获取数组中任意元素。 每个整数数组表示该维度的下标值。 当索引的元素个数就是目标 `ndarray` 的维度时，会变得相当直接。

以下示例获取了 `ndarray` 对象中每一行指定列的一个元素。 因此，行索引包含所有行号，列索引指定要选择的元素。

示例 1

```
import numpy as np

x = np.array([[1,  2],  [3,  4],  [5,  6]])
y = x[[0,1,2],  [0,1,0]]
print y
```

输出如下：

```
[1  4  5]
```

该结果包括数组中 `(0,0)` ， `(1,1)` 和 `(2,0)` 位置处的元素。

下面的示例获取了 4X3 数组中的每个角处的元素。 行索引是 `[0,0]` 和 `[3,3]` ，而列索引是 `[0,2]` 和 `[0,2]` 。

示例 2

```
import numpy as np
x = np.array([[  0,  1,  2],[  3,  4,  5],[  6,  7,  8],[  9,  1
0,  11]])
print  '我们的数组是：'
print x
print  '\n'
rows = np.array([[0,0],[3,3]])
cols = np.array([[0,2],[0,2]])
y = x[rows,cols]
print  '这个数组的每个角处的元素是：'
print y
```

输出如下：

```
我们的数组是：

[[ 0  1  2]

 [ 3  4  5]

 [ 6  7  8]

 [ 9 10 11]]

这个数组的每个角处的元素是：
[[ 0  2]

 [ 9 11]]
```

返回的结果是包含每个角元素的 `ndarray` 对象。

高级和基本索引可以通过使用切片 `:` 或省略号 `...` 与索引数组组合。 以下示例使用 `slice` 作为列索引和高级索引。 当切片用于两者时，结果是相同的。 但高级索引会导致复制，并且可能有不同的内存布局。

示例 3

```
import numpy as np
x = np.array([[  0,  1,  2],[  3,  4,  5],[  6,  7,  8],[  9,  1
0,  11]])
print  '我们的数组是：'
print x
print  '\n'
# 切片
z = x[1:4,1:3]
print  '切片之后，我们的数组变为：'
print z
print  '\n'
# 对列使用高级索引
y = x[1:4,[1,2]]
print  '对列使用高级索引来切片：'
print y
```

输出如下：

```
我们的数组是：
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]

切片之后，我们的数组变为：
[[ 4  5]
 [ 7  8]
 [10 11]]

对列使用高级索引来切片：
[[ 4  5]
 [ 7  8]
 [10 11]]
```

布尔索引

当结果对象是布尔运算（例如比较运算符）的结果时，将使用此类型的高级索引。

示例 1

这个例子中，大于 5 的元素会作为布尔索引的结果返回。

```
import numpy as np
x = np.array([[  0,  1,  2],[  3,  4,  5],[  6,  7,  8],[  9,  1
0,  11]])
print  '我们的数组是：'
print x
print  '\n'
# 现在我们会打印出大于 5 的元素
print  '大于 5 的元素是：'
print x[x >  5]
```

输出如下：

```
我们的数组是：
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]

大于 5 的元素是：
[ 6  7  8  9 10 11]
```

示例 2

这个例子使用了 `~` （取补运算符）来过滤 `NaN` 。

```
import numpy as np
a = np.array([np.nan,  1,2,np.nan,3,4,5])
print a[~np.isnan(a)]
```

输出如下：

```
[ 1.   2.   3.   4.   5.]
```

示例 3

以下示例显示如何从数组中过滤掉非复数元素。

```
import numpy as np
a = np.array([1,  2+6j,  5,  3.5+5j])
print a[np.iscomplex(a)]
```

输出如下：

```
[2.0+6.j  3.5+5.j]
```

NumPy - 广播

术语广播是指 NumPy 在算术运算期间处理不同形状的数组的能力。 对数组的算术运算通常在相应的元素上进行。 如果两个阵列具有完全相同的形状，则这些操作被无缝执行。

示例 1

```
import numpy as np

a = np.array([1,2,3,4])
b = np.array([10,20,30,40])
c = a * b
print c
```

输出如下：

```
[10   40   90   160]
```

如果两个数组的维数不相同，则元素到元素的操作是不可能的。 然而，在 NumPy 中仍然可以对形状不相似的数组进行操作，因为它拥有广播功能。 较小的数组会广播到较大数组的大小，以便使它们的形状可兼容。

如果满足以下规则，可以进行广播：

- `ndim` 较小的数组会在前面追加一个长度为 1 的维度。
- 输出数组的每个维度的大小是输入数组该维度大小的最大值。
- 如果输入在每个维度中的大小与输出大小匹配，或其值正好为 1，则在计算中可它。
- 如果输入的某个维度大小为 1，则该维度中的第一个数据元素将用于该维度的所有计算。

如果上述规则产生有效结果，并且满足以下条件之一，那么数组被称为可广播的。

- 数组拥有相同形状。
- 数组拥有相同的维数，每个维度拥有相同长度，或者长度为 1。
- 数组拥有极少的维度，可以在其前面追加长度为 1 的维度，使上述条件成立。

下面的例称展示了广播的示例。

示例 2

```
import numpy as np
a = np.array([[0.0,0.0,0.0],[10.0,10.0,10.0],[20.0,20.0,20.0],[3
0.0,30.0,30.0]])
b = np.array([1.0,2.0,3.0])
print  '第一个数组：'
print a
print  '\n'
print  '第二个数组：'
print b
print  '\n'
print  '第一个数组加第二个数组：'
print a + b
```

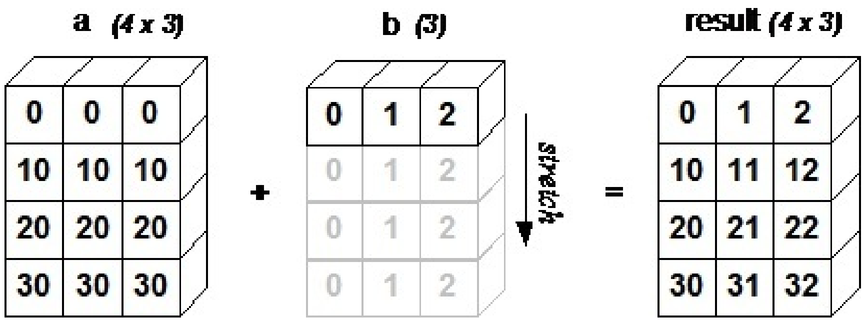
输出如下：

```
第一个数组：
[[ 0. 0. 0.]
 [ 10. 10. 10.]
 [ 20. 20. 20.]
 [ 30. 30. 30.]]

第二个数组：
[ 1. 2. 3.]

第一个数组加第二个数组：
[[ 1. 2. 3.]
 [ 11. 12. 13.]
 [ 21. 22. 23.]
 [ 31. 32. 33.]]
```

下面的图片展示了数组 `b` 如何通过广播来与数组 `a` 兼容。



NumPy - 数组上的迭代

NumPy 包包含一个迭代器对象 `numpy.nditer` 。 它是一个有效的多维迭代器对象，可以用于在数组上进行迭代。 数组的每个元素可使用 Python 的标准 `Iterator` 接口来访问。

让我们使用 `arange()` 函数创建一个 3X4 数组，并使用 `nditer` 对它进行迭代。

示例 1

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
print  '原始数组是：'
print a print  '\n'
print  '修改后的数组是：'
for x in np.nditer(a):
    print x,
```

输出如下：

```
原始数组是：
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

修改后的数组是：
0 5 10 15 20 25 30 35 40 45 50 55
```

示例 2

迭代的顺序匹配数组的内容布局，而不考虑特定的排序。 这可以通过迭代上述数组的转置来看到。

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
print  '原始数组是：'
print a
print  '\n'
print  '原始数组的转置是：'
b = a.T
print b
print  '\n'
print  '修改后的数组是：'
for x in np.nditer(b):
    print x,
```

输出如下：

```
原始数组是：
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

原始数组的转置是：
[[ 0 20 40]
 [ 5 25 45]
 [10 30 50]
 [15 35 55]]

修改后的数组是：
0 5 10 15 20 25 30 35 40 45 50 55
```

迭代顺序

如果相同元素使用 F 风格顺序存储，则迭代器选择以更有效的方式对数组进行迭代。

示例 1

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
print  '原始数组是：'
print a print  '\n'
print  '原始数组的转置是：'
b = a.T
print b
print  '\n'
print  '以 C 风格顺序排序：'
c = b.copy(order='C')
print c for x in np.nditer(c):
    print x,
print  '\n'
print  '以 F 风格顺序排序：'
c = b.copy(order='F')
print c
for x in np.nditer(c):
    print x,
```

输出如下：

```
原始数组是：
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

原始数组的转置是：
[[ 0 20 40]
 [ 5 25 45]
 [10 30 50]
 [15 35 55]]

以 C 风格顺序排序：
[[ 0 20 40]
 [ 5 25 45]
 [10 30 50]
 [15 35 55]]
0 20 40 5 25 45 10 30 50 15 35 55

以 F 风格顺序排序：
[[ 0 20 40]
 [ 5 25 45]
 [10 30 50]
 [15 35 55]]
0 5 10 15 20 25 30 35 40 45 50 55
```

示例 2

可以通过显式提醒，来强制 `nditer` 对象使用某种顺序：

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
print '原始数组是：'
print a
print '\n'
print '以 C 风格顺序排序：'
for x in np.nditer(a, order = 'C'):
    print x,
print '\n'
print '以 F 风格顺序排序：'
for x in np.nditer(a, order = 'F'):
    print x,
```

输出如下：

```
原始数组是：
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

以 C 风格顺序排序：
0 5 10 15 20 25 30 35 40 45 50 55

以 F 风格顺序排序：
0 20 40 5 25 45 10 30 50 15 35 55
```

修改数组的值

`nditer` 对象有另一个可选参数 `op_flags` 。其默认值为只读，但可以设置为读写或只写模式。这将允许使用此迭代器修改数组元素。

示例

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
print '原始数组是：'
print a
print '\n'
for x in np.nditer(a, op_flags=['readwrite']):
    x[...]=2*x
print '修改后的数组是：'
print a
```

输出如下：

```
原始数组是：
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

修改后的数组是：
[[ 0 10 20 30]
 [ 40 50 60 70]
 [ 80 90 100 110]]
```

外部循环

`nditer` 类的构造器拥有 `flags` 参数，它可以接受下列值：

序号	参数及描述
1.	`c_index` 可以跟踪 C 顺序的索引
2.	`f_index` 可以跟踪 Fortran 顺序的索引
3.	`multi-index` 每次迭代可以跟踪一种索引类型
4.	`external_loop` 给出的值是具有多个值的一维数组，而不是零维数组

示例

在下面的示例中，迭代器遍历对应于每列的一维数组。

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
print '原始数组是：'
print a
print '\n'
print '修改后的数组是：'
for x in np.nditer(a, flags = ['external_loop'], order = 'F'):

    print x,
```

输出如下：

```
原始数组是：
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

修改后的数组是：
[ 0 20 40] [ 5 25 45] [10 30 50] [15 35 55]
```

广播迭代

如果两个数组是可广播的，`nditer` 组合对象能够同时迭代它们。假设数组 `a` 具有维度 3X4，并且存在维度为 1X4 的另一个数组 `b`，则使用以下类型的迭代器（数组 `b` 被广播到 `a` 的大小）。

示例

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
print  '第一个数组：'
print a
print  '\n'
print  '第二个数组：'
b = np.array([1,  2,  3,  4], dtype =  int)
print b
print  '\n'
print  '修改后的数组是：'
for x,y in np.nditer([a,b]):
    print  "%d:%d"  %  (x,y),
```

输出如下：

```
第一个数组：
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

第二个数组：
[1 2 3 4]

修改后的数组是：
0:1 5:2 10:3 15:4 20:1 25:2 30:3 35:4 40:1 45:2 50:3 55:4
```

NumPy - 数组操作

NumPy包中有几个例程用于处理 `ndarray` 对象中的元素。 它们可以分为以下类型：

修改形状

序号	形状及描述
1.	`reshape` 不改变数据的条件下修改形状
2.	`flat` 数组上的一维迭代器
3.	`flatten` 返回折叠为一维的数组副本
4.	`ravel` 返回连续的展开数组

`numpy.reshape`

这个函数在不改变数据的条件下修改形状，它接受如下参数：

```
numpy.reshape(arr, newshape, order='C')
```

其中：

- `arr` ：要修改形状的数组
- `newshape` ：整数或者整数数组，新的形状应当兼容原有形状
- `order` ： `'C'` 为 C 风格顺序， `'F'` 为 F 风格顺序， `'A'` 为保留原顺序。

例子

```
import numpy as np
a = np.arange(8)
print '原始数组：'
print a
print '\n'

b = a.reshape(4,2)
print '修改后的数组：'
print b
```

输出如下：

```
原始数组：
[0 1 2 3 4 5 6 7]

修改后的数组：
[[0 1]
 [2 3]
 [4 5]
 [6 7]]
```

`numpy.ndarray.flat`

该函数返回数组上的一维迭代器，行为类似 Python 内建的迭代器。

例子

```
import numpy as np
a = np.arange(8).reshape(2,4)
print '原始数组：'
print a
print '\n'

print '调用 flat 函数之后：'
# 返回展开数组中的下标的对应元素
print a.flat[5]
```

输出如下：

```
原始数组：
[[0 1 2 3]
 [4 5 6 7]]

调用 flat 函数之后：
5
```

`numpy.ndarray.flatten`

该函数返回折叠为一维的数组副本，函数接受下列参数：

```
ndarray.flatten(order='C')
```

其中：

- `order`：`'C'` -- 按行，`'F'` -- 按列，`'A'` -- 原顺序，`'k'` -- 元素在内存中的出现顺序。

例子

```
import numpy as np
a = np.arange(8).reshape(2,4)

print '原数组：'
print a
print '\n'
# default is column-major

print '展开的数组：'
print a.flatten()
print '\n'

print '以 F 风格顺序展开的数组：'
print a.flatten(order = 'F')
```

输出如下：

```
原数组：
[[0 1 2 3]
 [4 5 6 7]]

展开的数组：
[0 1 2 3 4 5 6 7]

以 F 风格顺序展开的数组：
[0 4 1 5 2 6 3 7]
```

`numpy.ravel`

这个函数返回展开的一维数组，并且按需生成副本。返回的数组和输入数组拥有相同数据类型。这个函数接受两个参数。

```
numpy.ravel(a, order='C')
```

构造器接受下列参数：

- `order`：`'C'` -- 按行，`'F'` -- 按列，`'A'` -- 原顺序，`'k'` -- 元素在内存中的出现顺序。

例子

```
import numpy as np
a = np.arange(8).reshape(2,4)

print '原数组：'
print a
print '\n'

print '调用 ravel 函数之后：'
print a.ravel()
print '\n'

print '以 F 风格顺序调用 ravel 函数之后：'
print a.ravel(order = 'F')
```

```
原数组：
[[0 1 2 3]
 [4 5 6 7]]

调用 ravel 函数之后：
[0 1 2 3 4 5 6 7]

以 F 风格顺序调用 ravel 函数之后：
[0 4 1 5 2 6 3 7]
```

翻转操作

序号	操作及描述
1.	`transpose` 翻转数组的维度
2.	`ndarray.T` 和 `self.transpose()` 相同
3.	`rollaxis` 向后滚动指定的轴
4.	`swapaxes` 互换数组的两个轴

`numpy.transpose`

这个函数翻转给定数组的维度。如果可能的话它会返回一个视图。函数接受下列参数：

```
numpy.transpose(arr, axes)
```

其中：

- `arr` ：要转置的数组
- `axes` ：整数的列表，对应维度，通常所有维度都会翻转。

例子

```
import numpy as np
a = np.arange(12).reshape(3,4)

print '原数组：'
print a
print '\n'

print '转置数组：'
print np.transpose(a)
```

输出如下：

```
原数组：
[[ 0 1 2 3]
 [ 4 5 6 7]
 [ 8 9 10 11]]

转置数组：
[[ 0 4 8]
 [ 1 5 9]
 [ 2 6 10]
 [ 3 7 11]]
```

`numpy.ndarray.T`

该函数属于 `ndarray` 类，行为类似于 `numpy.transpose` 。

例子

```
import numpy as np
a = np.arange(12).reshape(3,4)

print '原数组：'
print a
print '\n'

print '转置数组：'
print a.T
```

输出如下：

```
原数组：
[[ 0 1 2 3]
 [ 4 5 6 7]
 [ 8 9 10 11]]

转置数组：
[[ 0 4 8]
 [ 1 5 9]
 [ 2 6 10]
 [ 3 7 11]]
```

`numpy.rollaxis`

该函数向后滚动特定的轴，直到一个特定位置。这个函数接受三个参数：

```
numpy.rollaxis(arr, axis, start)
```

其中：

- `arr`：输入数组
- `axis`：要向后滚动的轴，其它轴的相对位置不会改变
- `start`：默认为零，表示完整的滚动。会滚动到特定位置。

例子

```
# 创建了三维的 ndarray
import numpy as np
a = np.arange(8).reshape(2,2,2)

print '原数组：'
print a
print '\n'
# 将轴 2 滚动到轴 0（宽度到深度）

print '调用 rollaxis 函数：'
print np.rollaxis(a,2)
# 将轴 0 滚动到轴 1：（宽度到高度）
print '\n'

print '调用 rollaxis 函数：'
print np.rollaxis(a,2,1)
```

输出如下：

```
原数组：
[[[0 1]
 [2 3]]
 [[4 5]
 [6 7]]]

调用 rollaxis 函数：
[[[0 2]
 [4 6]]
 [[1 3]
 [5 7]]]

调用 rollaxis 函数：
[[[0 2]
 [1 3]]
 [[4 6]
 [5 7]]]
```

`numpy.swapaxes`

该函数交换数组的两个轴。对于 1.10 之前的 NumPy 版本，会返回交换后数组的视图。这个函数接受下列参数：

```
numpy.swapaxes(arr, axis1, axis2)
```

- `arr`：要交换其轴的输入数组
- `axis1`：对应第一个轴的整数
- `axis2`：对应第二个轴的整数

```
# 创建了三维的 ndarray
import numpy as np
a = np.arange(8).reshape(2,2,2)

print '原数组：'
print a
print '\n'
# 现在交换轴 0（深度方向）到轴 2（宽度方向）

print '调用 swapaxes 函数后的数组：'
print np.swapaxes(a, 2, 0)
```

输出如下：

```
原数组：
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]

调用 swapaxes 函数后的数组：
[[[0 4]
  [2 6]]

 [[1 5]
  [3 7]]]
```

修改维度

序号	维度和描述
1.	`broadcast` 产生模仿广播的对象
2.	`broadcast_to` 将数组广播到新形状
3.	`expand_dims` 扩展数组的形状
4.	`squeeze` 从数组的形状中删除单维条目

`broadcast`

如前所述，NumPy 已经内置了对广播的支持。 此功能模仿广播机制。 它返回一个对象，该对象封装了将一个数组广播到另一个数组的结果。

该函数使用两个数组作为输入参数。 下面的例子说明了它的用法。

```
import numpy as np
x = np.array([[1], [2], [3]])
y = np.array([4, 5, 6])

# 对 y 广播 x
b = np.broadcast(x,y)
# 它拥有 iterator 属性，基于自身组件的迭代器元组

print '对 y 广播 x：'
r,c = b.iters
print r.next(), c.next()
print r.next(), c.next()
print '\n'
# shape 属性返回广播对象的形状

print '广播对象的形状：'
print b.shape
print '\n'
# 手动使用 broadcast 将 x 与 y 相加
b = np.broadcast(x,y)
c = np.empty(b.shape)

print '手动使用 broadcast 将 x 与 y 相加：'
print c.shape
print '\n'
c.flat = [u + v for (u,v) in b]

print '调用 flat 函数：'
print c
print '\n'
# 获得了和 NumPy 内建的广播支持相同的结果

print 'x 与 y 的和：'
print x + y
```

输出如下：

```
对 y 广播 x：
1 4
1 5

广播对象的形状：
(3, 3)

手动使用 broadcast 将 x 与 y 相加：
(3, 3)

调用 flat 函数：
[[ 5. 6. 7.]
 [ 6. 7. 8.]
 [ 7. 8. 9.]]

x 与 y 的和：
[[5 6 7]
 [6 7 8]
 [7 8 9]]
```

numpy.broadcast_to

此函数将数组广播到新形状。 它在原始数组上返回只读视图。 它通常不连续。 如果新形状不符合 NumPy 的广播规则，该函数可能会抛出 `ValueError` 。

注意 - 此功能可用于 1.10.0 及以后的版本。

该函数接受以下参数。

```
numpy.broadcast_to(array, shape, subok)
```

例子

```
import numpy as np
a = np.arange(4).reshape(1,4)

print '原数组：'
print a
print '\n'

print '调用 broadcast_to 函数之后：'
print np.broadcast_to(a,(4,4))
```

输出如下：

```
[[0  1  2  3]
 [0  1  2  3]
 [0  1  2  3]
 [0  1  2  3]]
```

numpy.expand_dims

函数通过在指定位置插入新的轴来扩展数组形状。该函数需要两个参数：

```
numpy.expand_dims(arr, axis)
```

其中：

- `arr`：输入数组
- `axis`：新轴插入的位置

例子

```
import numpy as np
x = np.array(([1,2],[3,4]))

print '数组 x：'
print x
print '\n'
y = np.expand_dims(x, axis = 0)

print '数组 y：'
print y
print '\n'

print '数组 x 和 y 的形状：'
print x.shape, y.shape
print '\n'
# 在位置 1 插入轴
y = np.expand_dims(x, axis = 1)

print '在位置 1 插入轴之后的数组 y：'
print y
print '\n'

print 'x.ndim 和 y.ndim：'
print x.ndim,y.ndim
print '\n'

print 'x.shape 和 y.shape：'
print x.shape, y.shape
```

输出如下：

```
数组 x：
[[1 2]
 [3 4]]

数组 y：
[[[1 2]
 [3 4]]]

数组 x 和 y 的形状：
(2, 2) (1, 2, 2)

在位置 1 插入轴之后的数组 y：
[[[1 2]]
 [[3 4]]]

x.shape 和 y.shape：
2 3

x.shape and y.shape:
(2, 2) (2, 1, 2)
```

numpy.squeeze

函数从给定数组的形状中删除一维条目。 此函数需要两个参数。

```
numpy.squeeze(arr, axis)
```

其中：

- `arr`：输入数组
- `axis`：整数或整数元组，用于选择形状中单一维度条目的子集

例子

```
import numpy as np
x = np.arange(9).reshape(1,3,3)

print '数组 x：'
print x
print '\n'
y = np.squeeze(x)

print '数组 y：'
print y
print '\n'

print '数组 x 和 y 的形状：'
print x.shape, y.shape
```

输出如下：

```
数组 x：
[[[0 1 2]
  [3 4 5]
  [6 7 8]]]

数组 y：
[[0 1 2]
 [3 4 5]
 [6 7 8]]

数组 x 和 y 的形状：
(1, 3, 3) (3, 3)
```

数组的连接

序号	数组及描述
1.	`concatenate` 沿着现存的轴连接数据序列
2.	`stack` 沿着新轴连接数组序列
3.	`hstack` 水平堆叠序列中的数组（列方向）
4.	`vstack` 竖直堆叠序列中的数组（行方向）

`numpy.concatenate`

数组的连接是指连接。 此函数用于沿指定轴连接相同形状的两个或多个数组。 该函数接受以下参数。

```
numpy.concatenate((a1, a2, ...), axis)
```

其中：

- `a1, a2, ...`：相同类型的数组序列
- `axis`：沿着它连接数组的轴，默认为 0

例子

```
import numpy as np
a = np.array([[1,2],[3,4]])

print '第一个数组：'
print a
print '\n'
b = np.array([[5,6],[7,8]])

print '第二个数组：'
print b
print '\n'
# 两个数组的维度相同

print '沿轴 0 连接两个数组：'
print np.concatenate((a,b))
print '\n'

print '沿轴 1 连接两个数组：'
print np.concatenate((a,b),axis = 1)
```

输出如下：

```
第一个数组：
[[1 2]
 [3 4]]

第二个数组：
[[5 6]
 [7 8]]

沿轴 0 连接两个数组：
[[1 2]
 [3 4]
 [5 6]
 [7 8]]

沿轴 1 连接两个数组：
[[1 2 5 6]
 [3 4 7 8]]
```

`numpy.stack`

此函数沿新轴连接数组序列。 此功能添加自 NumPy 版本 1.10.0。 需要提供以下参数。

```
numpy.stack(arrays, axis)
```

其中：

- `arrays` ：相同形状的数组序列
- `axis` ：返回数组中的轴，输入数组沿着它来堆叠

```
import numpy as np
a = np.array([[1,2],[3,4]])

print '第一个数组：'
print a
print '\n'
b = np.array([[5,6],[7,8]])

print '第二个数组：'
print b
print '\n'

print '沿轴 0 堆叠两个数组：'
print np.stack((a,b),0)
print '\n'

print '沿轴 1 堆叠两个数组：'
print np.stack((a,b),1)
```

输出如下：

```
第一个数组：
[[1 2]
 [3 4]]

第二个数组：
[[5 6]
 [7 8]]

沿轴 0 堆叠两个数组：
[[[1 2]
 [3 4]]
 [[5 6]
 [7 8]]]

沿轴 1 堆叠两个数组：
[[[1 2]
 [5 6]]
 [[3 4]
 [7 8]]]
```

`numpy.hstack`

`numpy.stack` 函数的变体，通过堆叠来生成水平的单个数组。

例子

```
import numpy as np
a = np.array([[1,2],[3,4]])

print '第一个数组：'
print a
print '\n'
b = np.array([[5,6],[7,8]])

print '第二个数组：'
print b
print '\n'

print '水平堆叠：'
c = np.hstack((a,b))
print c
print '\n'
```

输出如下：

```
第一个数组：
[[1 2]
 [3 4]]

第二个数组：
[[5 6]
 [7 8]]

水平堆叠：
[[1 2 5 6]
 [3 4 7 8]]
```

numpy.vstack

`numpy.stack` 函数的变体，通过堆叠来生成竖直的单个数组。

```
import numpy as np
a = np.array([[1,2],[3,4]])

print '第一个数组：'
print a
print '\n'
b = np.array([[5,6],[7,8]])

print '第二个数组：'
print b
print '\n'

print '竖直堆叠：'
c = np.vstack((a,b))
print c
```

输出如下：

```
第一个数组：
[[1 2]
 [3 4]]

第二个数组：
[[5 6]
 [7 8]]

竖直堆叠：
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

数组分割

序号	数组及操作
1.	`split` 将一个数组分割为多个子数组
2.	`hsplit` 将一个数组水平分割为多个子数组（按列）
3.	`vsplit` 将一个数组竖直分割为多个子数组（按行）

`numpy.split`

该函数沿特定的轴将数组分割为子数组。函数接受三个参数：

```
numpy.split(ary, indices_or_sections, axis)
```

其中：

- `ary`：被分割的输入数组
- `indices_or_sections`：可以是整数，表明要从输入数组创建的，等大小的子数组的数量。 如果此参数是一维数组，则其元素表明要创建新子数组的点。
- `axis`：默认为 0

例子

```
import numpy as np
a = np.arange(9)

print '第一个数组：'
print a
print '\n'

print '将数组分为三个大小相等的子数组：'
b = np.split(a,3)
print b
print '\n'

print '将数组在一维数组中表明的位置分割：'
b = np.split(a,[4,7])
print b
```

输出如下：

```
第一个数组：
[0 1 2 3 4 5 6 7 8]

将数组分为三个大小相等的子数组：
[array([0, 1, 2]), array([3, 4, 5]), array([6, 7, 8])]

将数组在一维数组中表明的位置分割：
[array([0, 1, 2, 3]), array([4, 5, 6]), array([7, 8])]
```

`numpy.hsplit`

`numpy.hsplit` 是 `split()` 函数的特例，其中轴为 1 表示水平分割，无论输入数组的维度是什么。

```
import numpy as np
a = np.arange(16).reshape(4,4)

print '第一个数组：'
print a
print '\n'

print '水平分割：'
b = np.hsplit(a,2)
print b
print '\n'
```

输出：

```
第一个数组：
[[ 0 1 2 3]
 [ 4 5 6 7]
 [ 8 9 10 11]
 [12 13 14 15]]

水平分割：

[array([[ 0,  1],

       [ 4,  5],

       [ 8,  9],

       [12, 13]]), array([[ 2,  3],

       [ 6,  7],

       [10, 11],

       [14, 15]])]
```

numpy.vsplit

`numpy.vsplit` 是 `split()` 函数的特例，其中轴为 0 表示竖直分割，无论输入数组的维度是什么。下面的例子使之更清楚。

```
import numpy as np
a = np.arange(16).reshape(4,4)

print '第一个数组：'
print a
print '\n'

print '竖直分割：'
b = np.vsplit(a,2)
print b
```

输出如下：

```
第一个数组：
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]

竖直分割：

[array([[0, 1, 2, 3],

       [4, 5, 6, 7]]), array([[ 8,  9, 10, 11],

       [12, 13, 14, 15]])]
```

添加/删除元素

序号	元素及描述
1.	`resize` 返回指定形状的新数组
2.	`append` 将值添加到数组末尾
3.	`insert` 沿指定轴将值插入到指定下标之前
4.	`delete` 返回删掉某个轴的子数组的新数组
5.	`unique` 寻找数组内的唯一元素

`numpy.resize`

此函数返回指定大小的新数组。 如果新大小大于原始大小，则包含原始数组中的元素的重复副本。 该函数接受以下参数。

```
numpy.resize(arr, shape)
```

其中：

- `arr`：要修改大小的输入数组
- `shape`：返回数组的新形状

例子

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])

print '第一个数组：'
print a
print '\n'

print '第一个数组的形状：'
print a.shape
print '\n'
b = np.resize(a, (3,2))

print '第二个数组：'
print b
print '\n'

print '第二个数组的形状：'
print b.shape
print '\n'
# 要注意 a 的第一行在 b 中重复出现，因为尺寸变大了

print '修改第二个数组的大小：'
b = np.resize(a,(3,3))
print b
```

输出如下：

```
第一个数组：
[[1 2 3]
 [4 5 6]]

第一个数组的形状：
(2, 3)

第二个数组：
[[1 2]
 [3 4]
 [5 6]]

第二个数组的形状：
(3, 2)

修改第二个数组的大小：
[[1 2 3]
 [4 5 6]
 [1 2 3]]
```

`numpy.append`

此函数在输入数组的末尾添加值。 附加操作不是原地的，而是分配新的数组。 此外，输入数组的维度必须匹配否则将生成 `ValueError` 。

函数接受下列函数：

```
numpy.append(arr, values, axis)
```

其中：

- `arr` ：输入数组
- `values` ：要向 `arr` 添加的值，比如和 `arr` 形状相同（除了要添加的轴）
- `axis` ：沿着它完成操作的轴。如果没有提供，两个参数都会被展开。

例子

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])

print '第一个数组：'
print a
print '\n'

print '向数组添加元素：'
print np.append(a, [7,8,9])
print '\n'

print '沿轴 0 添加元素：'
print np.append(a, [[7,8,9]],axis = 0)
print '\n'

print '沿轴 1 添加元素：'
print np.append(a, [[5,5,5],[7,8,9]],axis = 1)
```

输出如下：

```
第一个数组：
[[1 2 3]
 [4 5 6]]

向数组添加元素：
[1 2 3 4 5 6 7 8 9]

沿轴 0 添加元素：
[[1 2 3]
 [4 5 6]
 [7 8 9]]

沿轴 1 添加元素：
[[1 2 3 5 5 5]
 [4 5 6 7 8 9]]
```

`numpy.insert`

此函数在给定索引之前，沿给定轴在输入数组中插入值。 如果值的类型转换为要插入，则它与输入数组不同。 插入没有原地的，函数会返回一个新数组。 此外，如果未提供轴，则输入数组会被展开。

`insert()` 函数接受以下参数：

```
numpy.insert(arr, obj, values, axis)
```

其中：

- `arr` ：输入数组
- `obj` ：在其之前插入值的索引
- `values` ：要插入的值
- `axis` ：沿着它插入的轴，如果未提供，则输入数组会被展开

例子

```
import numpy as np
a = np.array([[1,2],[3,4],[5,6]])

print '第一个数组：'
print a
print '\n'

print '未传递 Axis 参数。 在插入之前输入数组会被展开。'
print np.insert(a,3,[11,12])
print '\n'
print '传递了 Axis 参数。 会广播值数组来配输入数组。'

print '沿轴 0 广播：'
print np.insert(a,1,[11],axis = 0)
print '\n'

print '沿轴 1 广播：'
print np.insert(a,1,11,axis = 1)
```

输出如下：

```
第一个数组：
[[1 2]
 [3 4]
 [5 6]]


未传递 Axis 参数。 在插入之前输入数组会被展开。
[ 1  2  3 11 12  4  5  6]


传递了 Axis 参数。 会广播值数组来配输入数组。
沿轴 0 广播：
[[ 1  2]
 [11 11]
 [ 3  4]
 [ 5  6]]


沿轴 1 广播：
[[ 1 11  2]
 [ 3 11  4]
 [ 5 11  6]]
```

`numpy.delete`

此函数返回从输入数组中删除指定子数组的新数组。 与 `insert()` 函数的情况一样，如果未提供轴参数，则输入数组将展开。 该函数接受以下参数：

```
Numpy.delete(arr, obj, axis)
```

其中：

- `arr`：输入数组
- `obj`：可以被切片，整数或者整数数组，表明要从输入数组删除的子数组
- `axis`：沿着它删除给定子数组的轴，如果未提供，则输入数组会被展开

例子

```
import numpy as np
a = np.arange(12).reshape(3,4)

print '第一个数组：'
print a
print '\n'

print '未传递 Axis 参数。 在插入之前输入数组会被展开。'
print np.delete(a,5)
print '\n'

print '删除第二列：'
print np.delete(a,1,axis = 1)
print '\n'

print '包含从数组中删除的替代值的切片：'
a = np.array([1,2,3,4,5,6,7,8,9,10])
print np.delete(a, np.s_[::2])
```

输出如下：

```
第一个数组：
[[ 0 1 2 3]
 [ 4 5 6 7]
 [ 8 9 10 11]]

未传递 Axis 参数。 在插入之前输入数组会被展开。
[ 0 1 2 3 4 6 7 8 9 10 11]

删除第二列：
[[ 0 2 3]
 [ 4 6 7]
 [ 8 10 11]]

包含从数组中删除的替代值的切片：
[ 2 4 6 8 10]
```

`numpy.unique`

此函数返回输入数组中的去重元素数组。 该函数能够返回一个元组，包含去重数组和相关索引的数组。 索引的性质取决于函数调用中返回参数的类型。

```
numpy.unique(arr, return_index, return_inverse, return_counts)
```

其中：

- `arr`：输入数组，如果不是一维数组则会展开
- `return_index`：如果为 `true`，返回输入数组中的元素下标
- `return_inverse`：如果为 `true`，返回去重数组的下标，它可以用于重构输入数组
- `return_counts`：如果为 `true`，返回去重数组中的元素在原数组中的出现次数

例子

```
import numpy as np
a = np.array([5,2,6,2,7,5,6,8,2,9])

print '第一个数组：'
print a
print '\n'

print '第一个数组的去重值：'
u = np.unique(a)
print u
print '\n'

print '去重数组的索引数组：'
u,indices = np.unique(a, return_index = True)
print indices
print '\n'

print '我们可以看到每个和原数组下标对应的数值：'
print a
print '\n'

print '去重数组的下标：'
u,indices = np.unique(a,return_inverse = True)
print u
print '\n'

print '下标为：'
print indices
print '\n'

print '使用下标重构原数组：'
print u[indices]
print '\n'

print '返回去重元素的重复数量：'
u,indices = np.unique(a,return_counts = True)
print u
print indices
```

输出如下：

```
第一个数组：
[5 2 6 2 7 5 6 8 2 9]

第一个数组的去重值：
[2 5 6 7 8 9]

去重数组的索引数组：
[1 0 2 4 7 9]

我们可以看到每个和原数组下标对应的数值：
[5 2 6 2 7 5 6 8 2 9]

去重数组的下标：
[2 5 6 7 8 9]

下标为：
[1 0 2 0 3 1 2 4 0 5]

使用下标重构原数组：
[5 2 6 2 7 5 6 8 2 9]

返回唯一元素的重复数量：
[2 5 6 7 8 9]
 [3 2 2 1 1 1]
```

NumPy - 位操作

下面是 NumPy 包中可用的位操作函数。

序号	操作及描述
1.	`bitwise_and` 对数组元素执行位与操作
2.	`bitwise_or` 对数组元素执行位或操作
3.	`invert` 计算位非
4.	`left_shift` 向左移动二进制表示的位
5.	`right_shift` 向右移动二进制表示的位

`bitwise_and`

通过 `np.bitwise_and()` 函数对输入数组中的整数的二进制表示的相应位执行位与运算。

例子

```
import numpy as np
print '13 和 17 的二进制形式：'
a,b = 13,17
print bin(a), bin(b)
print '\n'

print '13 和 17 的位与：'
print np.bitwise_and(13, 17)
```

输出如下：

```
13 和 17 的二进制形式：
0b1101 0b10001

13 和 17 的位与：
1
```

你可以使用下表验证此输出。 考虑下面的位与真值表。

A	B	AND
1	1	1
1	0	0
0	1	0
0	0	0

		1	1	0	1
AND					
	1	0	0	0	1
result	0	0	0	0	1

`bitwise_or`

通过 `np.bitwise_or()` 函数对输入数组中的整数的二进制表示的相应位执行位或运算。

例子

```
import numpy as np
a,b = 13,17
print '13 和 17 的二进制形式：'
print bin(a), bin(b)

print '13 和 17 的位或：'
print np.bitwise_or(13, 17)
```

输出如下：

```
13 和 17 的二进制形式：
0b1101 0b10001

13 和 17 的位或：
29
```

你可以使用下表验证此输出。 考虑下面的位或真值表。

A	B	OR
1	1	1
1	0	1
0	1	1
0	0	0

		1	1	0	1
OR					
	1	0	0	0	1
result	1	1	1	0	1

`invert`

此函数计算输入数组中整数的位非结果。 对于有符号整数，返回补码。

例子

```
import numpy as np

print '13 的位反转，其中 ndarray 的 dtype 是 uint8：'
print np.invert(np.array([13], dtype = np.uint8))
print '\n'
# 比较 13 和 242 的二进制表示，我们发现了位的反转

print '13 的二进制表示：'
print np.binary_repr(13, width = 8)
print '\n'

print '242 的二进制表示：'
print np.binary_repr(242, width = 8)
```

输出如下：

```
13 的位反转，其中 ndarray 的 dtype 是 uint8：
[242]

13 的二进制表示：
00001101

242 的二进制表示：
11110010
```

请注意，`np.binary_repr()` 函数返回给定宽度中十进制数的二进制表示。

`left_shift`

`numpy.left shift()` 函数将数组元素的二进制表示中的位向左移动到指定位置，右侧附加相等数量的 0。

例如，

```
import numpy as np

print '将 10 左移两位：'
print np.left_shift(10,2)
print '\n'

print '10 的二进制表示：'
print np.binary_repr(10, width = 8)
print '\n'

print '40 的二进制表示：'
print np.binary_repr(40, width = 8)
#  '00001010' 中的两位移动到了左边，并在右边添加了两个 0。
```

输出如下：

```
将 10 左移两位：
40

10 的二进制表示：
00001010

40 的二进制表示：
00101000
```

`right_shift`

`numpy.right_shift()` 函数将数组元素的二进制表示中的位向右移动到指定位置，左侧附加相等数量的 0。

```
import numpy as np

print '将 40 右移两位：'
print np.right_shift(40,2)
print '\n'

print '40 的二进制表示：'
print np.binary_repr(40, width = 8)
print '\n'

print '10 的二进制表示：'
print np.binary_repr(10, width = 8)
#  '00001010' 中的两位移动到了右边，并在左边添加了两个 0。
```

输出如下：

```
将 40 右移两位：
10

40 的二进制表示：
00101000

10 的二进制表示：
00001010
```

NumPy - 字符串函数

以下函数用于对 `dtype` 为 `numpy.string_` 或 `numpy.unicode_` 的数组执行向量化字符串操作。 它们基于 Python 内置库中的标准字符串函数。

序号	函数及描述
1.	`add()` 返回两个 `str` 或 `Unicode` 数组的逐个字符串连接
2.	`multiply()` 返回按元素多重连接后的字符串
3.	`center()` 返回给定字符串的副本，其中元素位于特定字符串的中央
4.	`capitalize()` 返回给定字符串的副本，其中只有第一个字符串大写
5.	`title()` 返回字符串或 Unicode 的按元素标题转换版本
6.	`lower()` 返回一个数组，其元素转换为小写
7.	`upper()` 返回一个数组，其元素转换为大写
8.	`split()` 返回字符串中的单词列表，并使用分隔符来分割
9.	`splitlines()` 返回元素中的行列表，以换行符分割
10.	`strip()` 返回数组副本，其中元素移除了开头或者结尾处的特定字符
11.	`join()` 返回一个字符串，它是序列中字符串的连接
12.	`replace()` 返回字符串的副本，其中所有子字符串的出现位置都被新字符串取代
13.	`decode()` 按元素调用 `str.decode`
14.	`encode()` 按元素调用 `str.encode`

这些函数在字符数组类（ `numpy.char` ）中定义。 较旧的 Numarray 包包含 `chararray` 类。 `numpy.char` 类中的上述函数在执行向量化字符串操作时非常有用。

`numpy.char.add()`

函数执行按元素的字符串连接。

```
import numpy as np
print '连接两个字符串：'
print np.char.add(['hello'],[' xyz'])
print '\n'

print '连接示例：'
print np.char.add(['hello', 'hi'],[' abc', ' xyz'])
```

输出如下：

```
连接两个字符串：
['hello xyz']

连接示例：
['hello abc' 'hi xyz']
```

`numpy.char.multiply()`

这个函数执行多重连接。

```
import numpy as np
print np.char.multiply('Hello ',3)
```

输出如下：

```
Hello Hello Hello
```

`numpy.char.center()`

此函数返回所需宽度的数组，以便输入字符串位于中心，并使用 `fillchar` 在左侧和右侧进行填充。

```
import numpy as np
# np.char.center(arr, width,fillchar)
print np.char.center('hello', 20,fillchar = '*')
```

输出如下：

```
*******hello********
```

`numpy.char.capitalize()`

函数返回字符串的副本，其中第一个字母大写

```
import numpy as np
print np.char.capitalize('hello world')
```

输出如下：

```
Hello world
```

`numpy.char.title()`

返回输入字符串的按元素标题转换版本，其中每个单词的首字母都大写。

```
import numpy as np
print np.char.title('hello how are you?')
```

输出如下：

```
Hello How Are You?
```

`numpy.char.lower()`

函数返回一个数组，其元素转换为小写。它对每个元素调用 `str.lower` 。

```
import numpy as np
print np.char.lower(['HELLO','WORLD'])
print np.char.lower('HELLO')
```

输出如下：

```
['hello' 'world']
hello
```

`numpy.char.upper()`

函数返回一个数组，其元素转换为大写。它对每个元素调用 `str.upper` 。

```
import numpy as np
print np.char.upper('hello')
print np.char.upper(['hello','world'])
```

输出如下：

```
HELLO
['HELLO' 'WORLD']
```

numpy.char.split()

此函数返回输入字符串中的单词列表。 默认情况下，空格用作分隔符。 否则，指定的分隔符字符用于分割字符串。

```
import numpy as np
print np.char.split ('hello how are you?')
print np.char.split ('TutorialsPoint,Hyderabad,Telangana', sep =
 ',')
```

输出如下：

```
['hello', 'how', 'are', 'you?']
['TutorialsPoint', 'Hyderabad', 'Telangana']
```

numpy.char.splitlines()

函数返回数组中元素的单词列表，以换行符分割。

```
import numpy as np
print np.char.splitlines('hello\nhow are you?')
print np.char.splitlines('hello\rhow are you?')
```

输出如下：

```
['hello', 'how are you?']
['hello', 'how are you?']
```

`'\n'` ， `'\r'` ， `'\r\n'` 都会用作换行符。

`numpy.char.strip()`

函数返回数组的副本，其中元素移除了开头或结尾处的特定字符。

```
import numpy as np
print np.char.strip('ashok arora','a')
print np.char.strip(['arora','admin','java'],'a')
```

输出如下：

```
shok aror
['ror' 'dmin' 'jav']
```

`numpy.char.join()`

这个函数返回一个字符串，其中单个字符由特定的分隔符连接。

```
import numpy as np
print np.char.join(':','dmy')
print np.char.join([':','-'],['dmy','ymd'])
```

输出如下：

```
d:m:y
['d:m:y' 'y-m-d']
```

`numpy.char.replace()`

这个函数返回字符串副本，其中所有字符序列的出现位置都被另一个给定的字符序列取代。

```
import numpy as np
print np.char.replace ('He is a good boy', 'is', 'was')
```

输出如下：

```
He was a good boy
```

`numpy.char.decode()`

这个函数在给定的字符串中使用特定编码调用 `str.decode()` 。

```
import numpy as np

a = np.char.encode('hello', 'cp500')
print a
print np.char.decode(a,'cp500')
```

输出如下：

```
\x88\x85\x93\x93\x96
hello
```

`numpy.char.encode()`

此函数对数组中的每个元素调用 `str.encode` 函数。 默认编码是 `utf_8` ，可以使用标准 Python 库中的编解码器。

```
import numpy as np
a = np.char.encode('hello', 'cp500')
print a
```

输出如下：

```
\x88\x85\x93\x93\x96
```

NumPy - 算数函数

很容易理解的是，NumPy 包含大量的各种数学运算功能。 NumPy 提供标准的三角函数，算术运算的函数，复数处理函数等。

三角函数

NumPy 拥有标准的三角函数，它为弧度制单位的给定角度返回三角函数比值。

示例

```
import numpy as np
a = np.array([0,30,45,60,90])
print  '不同角度的正弦值：'
# 通过乘 pi/180 转化为弧度
print np.sin(a*np.pi/180)
print  '\n'
print  '数组中角度的余弦值：'
print np.cos(a*np.pi/180)
print  '\n'
print  '数组中角度的正切值：'
print np.tan(a*np.pi/180)
```

输出如下：

```
不同角度的正弦值：

[ 0.          0.5         0.70710678  0.8660254   1.        ]


数组中角度的余弦值：
[  1.00000000e+00   8.66025404e-01   7.07106781e-01   5.00000000
e-01
   6.12323400e-17]


数组中角度的正切值：
[  0.00000000e+00   5.77350269e-01   1.00000000e+00   1.73205081
e+00
   1.63312394e+16]
```

`arcsin` ， `arccos` ，和 `arctan` 函数返回给定角度的 `sin` ， `cos` 和 `tan` 的反三角函数。 这些函数的结果可以通过 `numpy.degrees()` 函数通过将弧度制转换为角度制来验证。

示例

```
import numpy as np
a = np.array([0,30,45,60,90])
print '含有正弦值的数组：'
sin = np.sin(a*np.pi/180)
print sin
print '\n'
print '计算角度的反正弦，返回值以弧度为单位：'
inv = np.arcsin(sin)
print inv
print '\n'
print '通过转化为角度制来检查结果：'
print np.degrees(inv)
print '\n'
print 'arccos 和 arctan 函数行为类似：'
cos = np.cos(a*np.pi/180)
print cos
print '\n'
print '反余弦：'
inv = np.arccos(cos)
print inv
print '\n'
print '角度制单位：'
print np.degrees(inv)
print '\n'
print 'tan 函数：'
tan = np.tan(a*np.pi/180)
print tan
print '\n'
print '反正切：'
inv = np.arctan(tan)
print inv
print '\n'
print '角度制单位：'
print np.degrees(inv)
```

输出如下：

```
含有正弦值的数组：
[ 0.          0.5         0.70710678  0.8660254   1.        ]

计算角度的反正弦，返回值以弧度制为单位：
[ 0.          0.52359878  0.78539816  1.04719755  1.57079633]

通过转化为角度制来检查结果：
[  0.  30.  45.  60.  90.]

arccos 和 arctan 函数行为类似：
[  1.00000000e+00   8.66025404e-01   7.07106781e-01   5.00000000
e-01
   6.12323400e-17]

反余弦：
[ 0.          0.52359878  0.78539816  1.04719755  1.57079633]

角度制单位：
[  0.  30.  45.  60.  90.]

tan 函数：
[  0.00000000e+00   5.77350269e-01   1.00000000e+00   1.73205081
e+00
   1.63312394e+16]

反正切：
[ 0.          0.52359878  0.78539816  1.04719755  1.57079633]

角度制单位：
[  0.  30.  45.  60.  90.]
```

舍入函数

`numpy.around()`

这个函数返回四舍五入到所需精度的值。 该函数接受以下参数。

```
numpy.around(a,decimals)
```

其中：

序号	参数及描述
1.	`a` 输入数组
2.	`decimals` 要舍入的小数位数。 默认值为0。 如果为负，整数将四舍五入到小数点左侧的位置

示例

```
import numpy as np
a = np.array([1.0,5.55,  123,  0.567,  25.532])
print  '原数组：'
print a
print  '\n'
print  '舍入后：'
print np.around(a)
print np.around(a, decimals =  1)
print np.around(a, decimals =  -1)
```

输出如下：

```
原数组：

[   1.       5.55   123.       0.567   25.532]

舍入后：
[   1.    6.   123.    1.   26. ]

[   1.    5.6  123.    0.6  25.5]

[   0.    10.  120.    0.   30. ]
```

numpy.floor()

此函数返回不大于输入参数的最大整数。 即标量 `x` 的下限是最大的整数 `i` ，使得 `i <= x` 。 注意在Python中，向下取整总是从 0 舍入。

示例

```
import numpy as np
a = np.array([-1.7,  1.5,  -0.2,  0.6,  10])
print  '提供的数组：'
print a
print  '\n'
print  '修改后的数组：'
print np.floor(a)
```

输出如下：

```
提供的数组：

[ -1.7   1.5  -0.2   0.6  10. ]

修改后的数组：

[ -2.   1.  -1.   0.  10.]
```

numpy.ceil()

`ceil()` 函数返回输入值的上限，即，标量 `x` 的上限是最小的整数 `i` ，使得 `i> = x` 。

示例

```
import numpy as np
a = np.array([-1.7,  1.5,  -0.2,  0.6,  10])
print  '提供的数组：'
print a
print  '\n'
print  '修改后的数组：'
print np.ceil(a)
```

输出如下：

```
提供的数组：
[ -1.7   1.5  -0.2   0.6  10. ]

修改后的数组：
[ -1.   2.  -0.   1.  10.]
```

NumPy - 算数运算

用于执行算术运算（如 `add()` ，`subtract()` ，`multiply()` 和 `divide()` ）的输入数组必须具有相同的形状或符合数组广播规则。

示例

```
import numpy as np
a = np.arange(9, dtype = np.float_).reshape(3,3)
print '第一个数组：'
print a
print '\n'
print '第二个数组：'
b = np.array([10,10,10])
print b
print '\n'
print '两个数组相加：'
print np.add(a,b)
print '\n'
print '两个数组相减：'
print np.subtract(a,b)
print '\n'
print '两个数组相乘：'
print np.multiply(a,b)
print '\n'
print '两个数组相除：'
print np.divide(a,b)
```

输出如下：

```
第一个数组：
[[ 0.  1.  2.]
 [ 3.  4.  5.]
 [ 6.  7.  8.]]

第二个数组：
[10 10 10]

两个数组相加：
[[ 10.  11.  12.]
 [ 13.  14.  15.]
 [ 16.  17.  18.]]

两个数组相减：
[[-10.  -9.  -8.]
 [ -7.  -6.  -5.]
 [ -4.  -3.  -2.]]

两个数组相乘：
[[  0.  10.  20.]
 [ 30.  40.  50.]
 [ 60.  70.  80.]]

两个数组相除：
[[ 0.   0.1  0.2]
 [ 0.3  0.4  0.5]
 [ 0.6  0.7  0.8]]
```

让我们现在来讨论 NumPy 中提供的一些其他重要的算术函数。

`numpy.reciprocal()`

此函数返回参数逐元素的倒数，。 由于 Python 处理整数除法的方式，对于绝对值大于 1 的整数元素，结果始终为 0， 对于整数 0，则发出溢出警告。

示例

```
import numpy as np
a = np.array([0.25,  1.33,  1,  0,  100])
print '我们的数组是：'
print a
print '\n'
print '调用 reciprocal 函数：'
print np.reciprocal(a)
print '\n'
b = np.array([100], dtype =  int)
print '第二个数组：'
print b
print '\n'
print '调用 reciprocal 函数：'
print np.reciprocal(b)
```

输出如下：

```
我们的数组是：

[   0.25    1.33    1.      0.    100.  ]


调用 reciprocal 函数：
main.py:9: RuntimeWarning: divide by zero encountered in recipro
cal
  print np.reciprocal(a)

[ 4.         0.7518797  1.               inf  0.01     ]


第二个数组：
[100]


调用 reciprocal 函数：
[0]
```

numpy.power()

此函数将第一个输入数组中的元素作为底数，计算它与第二个输入数组中相应元素的幂。

```
import numpy as np
a = np.array([10,100,1000])
print '我们的数组是；'
print a
print '\n'
print '调用 power 函数：'
print np.power(a,2)
print '\n'
print '第二个数组：'
b = np.array([1,2,3])
print b
print '\n'
print '再次调用 power 函数：'
print np.power(a,b)
```

输出如下：

```
我们的数组是；

[  10  100 1000]


调用 power 函数：

[    100   10000 1000000]


第二个数组：

[1 2 3]


再次调用 power 函数：

[        10      10000 1000000000]
```

`numpy.mod()`

此函数返回输入数组中相应元素的除法余数。 函数 `numpy.remainder()` 也产生相同的结果。

```
import numpy as np
a = np.array([10,20,30])
b = np.array([3,5,7])
print '第一个数组：'
print a
print '\n'
print '第二个数组：'
print b
print '\n'
print '调用 mod() 函数：'
print np.mod(a,b)
print '\n'
print '调用 remainder() 函数：'
print np.remainder(a,b)
```

输出如下：

```
第一个数组：
[10 20 30]

第二个数组：
[3 5 7]

调用 mod() 函数：

[1 0 2]

调用 remainder() 函数：

[1 0 2]
```

以下函数用于对含有复数的数组执行操作。

- `numpy.real()` 返回复数类型参数的实部。
- `numpy.imag()` 返回复数类型参数的虚部。
- `numpy.conj()` 返回通过改变虚部的符号而获得的共轭复数。
- `numpy.angle()` 返回复数参数的角度。函数的参数是 `degree` 。如果为 `true` ，返回的角度以角度制来表示，否则为以弧度制来表示。

```
import numpy as np
a = np.array([-5.6j,  0.2j,  11.  ,  1+1j])
print '我们的数组是：'
print a
print '\n'
print '调用 real() 函数：'
print np.real(a)
print '\n'
print '调用 imag() 函数：'
print np.imag(a)
print '\n'
print '调用 conj() 函数：'
print np.conj(a)
print '\n'
print '调用 angle() 函数：'
print np.angle(a)
print '\n'
print '再次调用 angle() 函数（以角度制返回）：'
print np.angle(a, deg =  True)
```

输出如下：

```
我们的数组是：
[ 0.-5.6j 0.+0.2j 11.+0.j 1.+1.j ]

调用 real() 函数：
[ 0. 0. 11. 1.]

调用 imag() 函数：
[-5.6 0.2 0. 1. ]

调用 conj() 函数：
[ 0.+5.6j 0.-0.2j 11.-0.j 1.-1.j ]

调用 angle() 函数：
[-1.57079633 1.57079633 0. 0.78539816]

再次调用 angle() 函数（以角度制返回）：
[-90. 90. 0. 45.]
```

NumPy - 统计函数

NumPy 有很多有用的统计函数，用于从数组中给定的元素中查找最小，最大，百分标准差和方差等。 函数说明如下：

`numpy.amin()` 和 `numpy.amax()`

这些函数从给定数组中的元素沿指定轴返回最小值和最大值。

示例

```
import numpy as np
a = np.array([[3,7,5],[8,4,3],[2,4,9]])
print  '我们的数组是：'
print a
print  '\n'
print  '调用 amin() 函数：'
print np.amin(a,1)
print  '\n'
print  '再次调用 amin() 函数：'
print np.amin(a,0)
print  '\n'
print  '调用 amax() 函数：'
print np.amax(a)
print  '\n'
print  '再次调用 amax() 函数：'
print np.amax(a, axis =  0)
```

输出如下：

```
我们的数组是：
[[3 7 5]
[8 4 3]
[2 4 9]]

调用 amin() 函数：
[3 3 2]

再次调用 amin() 函数：
[2 4 3]

调用 amax() 函数：
9

再次调用 amax() 函数：
[8 7 9]
```

numpy.ptp()

`numpy.ptp()` 函数返回沿轴的值的范围（最大值 - 最小值）。

```
import numpy as np
a = np.array([[3,7,5],[8,4,3],[2,4,9]])
print '我们的数组是：'
print a
print '\n'
print '调用 ptp() 函数：'
print np.ptp(a)
print '\n'
print '沿轴 1 调用 ptp() 函数：'
print np.ptp(a, axis = 1)
print '\n'
print '沿轴 0 调用 ptp() 函数：'
print np.ptp(a, axis = 0)
```

输出如下：

```
我们的数组是：
[[3 7 5]
[8 4 3]
[2 4 9]]

调用 ptp() 函数：
7

沿轴 1 调用 ptp() 函数：
[4 5 7]

沿轴 0 调用 ptp() 函数：
[6 3 6]
```

`numpy.percentile()`

百分位数是统计中使用的度量，表示小于这个值的观察值的百分比。 函数 `numpy.percentile()` 接受以下参数。

```
numpy.percentile(a, q, axis)
```

其中：

序号	参数及描述
1.	`a` 输入数组
2.	`q` 要计算的百分位数，在 0 ~ 100 之间
3.	`axis` 沿着它计算百分位数的轴

示例

```
import numpy as np
a = np.array([[30,40,70],[80,20,10],[50,90,60]])
print '我们的数组是：'
print a
print '\n'
print '调用 percentile() 函数：'
print np.percentile(a,50)
print '\n'
print '沿轴 1 调用 percentile() 函数：'
print np.percentile(a,50, axis = 1)
print '\n'
print '沿轴 0 调用 percentile() 函数：'
print np.percentile(a,50, axis = 0)
```

输出如下：

```
我们的数组是：
[[30 40 70]
 [80 20 10]
 [50 90 60]]

调用 percentile() 函数：
50.0

沿轴 1 调用 percentile() 函数：
[ 40. 20. 60.]

沿轴 0 调用 percentile() 函数：
[ 50. 40. 60.]
```

numpy.median()

中值定义为将数据样本的上半部分与下半部分分开的值。 `numpy.median()` 函数的用法如下面的程序所示。

示例

```
import numpy as np
a = np.array([[30,65,70],[80,95,10],[50,90,60]])
print '我们的数组是：'
print a
print '\n'
print '调用 median() 函数：'
print np.median(a)
print '\n'
print '沿轴 0 调用 median() 函数：'
print np.median(a, axis = 0)
print '\n'
print '沿轴 1 调用 median() 函数：'
print np.median(a, axis = 1)
```

输出如下：

```
我们的数组是：
[[30 65 70]
 [80 95 10]
 [50 90 60]]

调用 median() 函数：
65.0

沿轴 0 调用 median() 函数：
[ 50. 90. 60.]

沿轴 1 调用 median() 函数：
[ 65. 80. 60.]
```

`numpy.mean()`

算术平均值是沿轴的元素的总和除以元素的数量。 `numpy.mean()` 函数返回数组中元素的算术平均值。 如果提供了轴，则沿其计算。

示例

```
import numpy as np
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
print  '我们的数组是：'
print a
print  '\n'
print  '调用 mean() 函数：'
print np.mean(a)
print  '\n'
print  '沿轴 0 调用 mean() 函数：'
print np.mean(a, axis =  0)
print  '\n'
print  '沿轴 1 调用 mean() 函数：'
print np.mean(a, axis =  1)
```

输出如下：

```
我们的数组是：
[[1 2 3]
 [3 4 5]
 [4 5 6]]

调用 mean() 函数：
3.66666666667

沿轴 0 调用 mean() 函数：
[ 2.66666667 3.66666667 4.66666667]

沿轴 1 调用 mean() 函数：
[ 2. 4. 5.]
```

`numpy.average()`

加权平均值是由每个分量乘以反映其重要性的因子得到的平均值。 `numpy.average()` 函数根据在另一个数组中给出的各自的权重计算数组中元素的加权平均值。 该函数可以接受一个轴参数。 如果没有指定轴，则数组会被展开。

考虑数组 `[1,2,3,4]` 和相应的权重 `[4,3,2,1]` ，通过将相应元素的乘积相加，并将和除以权重的和，来计算加权平均值。

```
加权平均值 = (1*4+2*3+3*2+4*1)/(4+3+2+1)
```

示例

```
import numpy as np
a = np.array([1,2,3,4])
print  '我们的数组是：'
print a
print  '\n'
print  '调用 average() 函数：'
print np.average(a)
print  '\n'
# 不指定权重时相当于 mean 函数
wts = np.array([4,3,2,1])
print  '再次调用 average() 函数：'
print np.average(a,weights = wts)
print  '\n'
# 如果 returned 参数设为 true，则返回权重的和
print  '权重的和：'
print np.average([1,2,3,  4],weights =  [4,3,2,1], returned =  T
rue)
```

输出如下：

```
我们的数组是：
[1 2 3 4]

调用 average() 函数：
2.5

再次调用 average() 函数：
2.0

权重的和：
(2.0, 10.0)
```

在多维数组中，可以指定用于计算的轴。

示例

```
import numpy as np
a = np.arange(6).reshape(3,2)
print '我们的数组是：'
print a
print '\n'
print '修改后的数组：'
wt = np.array([3,5])
print np.average(a, axis = 1, weights = wt)
print '\n'
print '修改后的数组：'
print np.average(a, axis = 1, weights = wt, returned = True)
```

输出如下：

```
我们的数组是：
[[0 1]
 [2 3]
 [4 5]]

修改后的数组：
[ 0.625 2.625 4.625]

修改后的数组：
(array([ 0.625, 2.625, 4.625]), array([ 8., 8., 8.]))
```

标准差

标准差是与均值的偏差的平方的平均值的平方根。 标准差公式如下：

```
std = sqrt(mean((x - x.mean())**2))
```

如果数组是 `[1，2，3，4]` ，则其平均值为 `2.5` 。 因此，差的平方是 `[2.25,0.25,0.25,2.25]` ，并且其平均值的平方根除以4，即 `sqrt(5/4)` 是 `1.1180339887498949` 。

示例

```
import numpy as np
print np.std([1,2,3,4])
```

输出如下：

```
1.1180339887498949
```

方差

方差是偏差的平方的平均值，即 `mean((x - x.mean())** 2)` 。 换句话说，标准差是方差的平方根。

示例

```
import numpy as np
print np.var([1,2,3,4])
```

输出如下：

```
1.25
```

NumPy - 排序、搜索和计数函数

NumPy中提供了各种排序相关功能。 这些排序函数实现不同的排序算法，每个排序算法的特征在于执行速度，最坏情况性能，所需的工作空间和算法的稳定性。 下表显示了三种排序算法的比较。

种类	速度	最坏情况	工作空间	稳定性
`'quicksort'`（快速排序）	1	`O(n^2)`	0	否
`'mergesort'`（归并排序）	2	`O(n*log(n))`	~n/2	是
`'heapsort'`（堆排序）	3	`O(n*log(n))`	0	否

`numpy.sort()`

`sort()` 函数返回输入数组的排序副本。 它有以下参数：

```
numpy.sort(a, axis, kind, order)
```

其中：

序号	参数及描述
1.	`a` 要排序的数组
2.	`axis` 沿着它排序数组的轴，如果没有数组会被展开，沿着最后的轴排序
3.	`kind` 默认为 `'quicksort'`（快速排序）
4.	`order` 如果数组包含字段，则是要排序的字段

示例

```
import numpy as np
a = np.array([[3,7],[9,1]])
print  '我们的数组是：'
print a
print  '\n'
print  '调用 sort() 函数：'
print np.sort(a)
print  '\n'
print  '沿轴 0 排序：'
print np.sort(a, axis =  0)
print  '\n'
# 在 sort 函数中排序字段
dt = np.dtype([('name',  'S10'),('age',  int)])
a = np.array([("raju",21),("anil",25),("ravi",  17),  ("amar",27
)], dtype = dt)
print  '我们的数组是：'
print a
print  '\n'
print  '按 name 排序：'
print np.sort(a, order =  'name')
```

输出如下：

```
我们的数组是：
[[3 7]
 [9 1]]

调用 sort() 函数：
[[3 7]
 [1 9]]

沿轴 0 排序：
[[3 1]
 [9 7]]

我们的数组是：
[('raju', 21) ('anil', 25) ('ravi', 17) ('amar', 27)]

按 name 排序：
[('amar', 27) ('anil', 25) ('raju', 21) ('ravi', 17)]
```

`numpy.argsort()`

`numpy.argsort()` 函数对输入数组沿给定轴执行间接排序，并使用指定排序类型返回数据的索引数组。 这个索引数组用于构造排序后的数组。

示例

```
import numpy as np
x = np.array([3,  1,  2])
print  '我们的数组是：'
print x
print  '\n'
print  '对 x 调用 argsort() 函数：'
y = np.argsort(x)
print y
print  '\n'
print  '以排序后的顺序重构原数组：'
print x[y]
print  '\n'
print  '使用循环重构原数组：'
for i in y:
    print x[i],
```

输出如下：

```
我们的数组是：
[3 1 2]

对 x 调用 argsort() 函数：
[1 2 0]

以排序后的顺序重构原数组：
[1 2 3]

使用循环重构原数组：
1 2 3
```

numpy.lexsort()

函数使用键序列执行间接排序。 键可以看作是电子表格中的一列。 该函数返回一个索引数组，使用它可以获得排序数据。 注意，最后一个键恰好是 sort 的主键。

示例

```
import numpy as np

nm =  ('raju','anil','ravi','amar')
dv =  ('f.y.',  's.y.',  's.y.',  'f.y.')
ind = np.lexsort((dv,nm))
print  '调用 lexsort() 函数：'
print ind
print  '\n'
print  '使用这个索引来获取排序后的数据：'
print  [nm[i]  +  ", "  + dv[i]  for i in ind]
```

输出如下：

```
调用 lexsort() 函数：
[3 1 0 2]

使用这个索引来获取排序后的数据：
['amar, f.y.', 'anil, s.y.', 'raju, f.y.', 'ravi, s.y.']
```

NumPy 模块有一些用于在数组内搜索的函数。 提供了用于找到最大值，最小值以及满足给定条件的元素的函数。

`numpy.argmax()` 和 `numpy.argmin()`

这两个函数分别沿给定轴返回最大和最小元素的索引。

示例

```
import numpy as np
a = np.array([[30,40,70],[80,20,10],[50,90,60]])
print  '我们的数组是：'
print a
print  '\n'
print  '调用 argmax() 函数：'
print np.argmax(a)
print  '\n'
print  '展开数组：'
print a.flatten()
print  '\n'
print  '沿轴 0 的最大值索引：'
maxindex = np.argmax(a, axis =  0)
print maxindex
print  '\n'
print  '沿轴 1 的最大值索引：'
maxindex = np.argmax(a, axis =  1)
print maxindex
print  '\n'
print  '调用 argmin() 函数：'
minindex = np.argmin(a)
print minindex
print  '\n'
print  '展开数组中的最小值：'
print a.flatten()[minindex]
print  '\n'
print  '沿轴 0 的最小值索引：'
minindex = np.argmin(a, axis =  0)
print minindex
print  '\n'
print  '沿轴 1 的最小值索引：'
minindex = np.argmin(a, axis =  1)
print minindex
```

输出如下：

```
我们的数组是：
[[30 40 70]
 [80 20 10]
 [50 90 60]]

调用 argmax() 函数：
7

展开数组：
[30 40 70 80 20 10 50 90 60]

沿轴 0 的最大值索引：
[1 2 0]

沿轴 1 的最大值索引：
[2 0 1]

调用 argmin() 函数：
5

展开数组中的最小值：
10

沿轴 0 的最小值索引：
[0 1 1]

沿轴 1 的最小值索引：
[0 2 0]
```

`numpy.nonzero()`

`numpy.nonzero()` 函数返回输入数组中非零元素的索引。

示例

```
import numpy as np
a = np.array([[30,40,0],[0,20,10],[50,0,60]])
print  '我们的数组是：'
print a
print  '\n'
print  '调用 nonzero() 函数：'
print np.nonzero (a)
```

输出如下：

```
我们的数组是：
[[30 40 0]
 [ 0 20 10]
 [50 0 60]]

调用 nonzero() 函数：
(array([0, 0, 1, 1, 2, 2]), array([0, 1, 1, 2, 0, 2]))
```

numpy.where()

`where()` 函数返回输入数组中满足给定条件的元素的索引。

示例

```
import numpy as np
x = np.arange(9.).reshape(3,  3)
print  '我们的数组是：'
print x
print  '大于 3 的元素的索引：'
y = np.where(x >  3)
print y
print  '使用这些索引来获取满足条件的元素：'
print x[y]
```

输出如下：

```
我们的数组是：
[[ 0. 1. 2.]
 [ 3. 4. 5.]
 [ 6. 7. 8.]]

大于 3 的元素的索引：
(array([1, 1, 2, 2, 2]), array([1, 2, 0, 1, 2]))

使用这些索引来获取满足条件的元素：
[ 4. 5. 6. 7. 8.]
```

numpy.extract()

`extract()` 函数返回满足任何条件的元素。

```
import numpy as np
x = np.arange(9.).reshape(3,  3)
print  '我们的数组是：'
print x
# 定义条件
condition = np.mod(x,2)  ==  0
print  '按元素的条件值：'
print condition
print  '使用条件提取元素：'
print np.extract(condition, x)
```

输出如下：

```
我们的数组是：
[[ 0. 1. 2.]
 [ 3. 4. 5.]
 [ 6. 7. 8.]]

按元素的条件值：
[[ True False True]
 [False True False]
 [ True False True]]

使用条件提取元素：
[ 0. 2. 4. 6. 8.]
```

NumPy - 字节交换

我们已经看到，存储在计算机内存中的数据取决于 CPU 使用的架构。 它可以是小端（最小有效位存储在最小地址中）或大端（最小有效字节存储在最大地址中）。

`numpy.ndarray.byteswap()`

`numpy.ndarray.byteswap()` 函数在两个表示：大端和小端之间切换。

```
import numpy as np
a = np.array([1,  256,  8755], dtype = np.int16)
print  '我们的数组是：'
print a
print  '以十六进制表示内存中的数据：'
print map(hex,a)
# byteswap() 函数通过传入 true 来原地交换
print  '调用 byteswap() 函数：'
print a.byteswap(True)
print  '十六进制形式：'
print map(hex,a)
# 我们可以看到字节已经交换了
```

输出如下：

```
我们的数组是：
[1 256 8755]

以十六进制表示内存中的数据：
['0x1', '0x100', '0x2233']

调用 byteswap() 函数：
[256 1 13090]

十六进制形式：
['0x100', '0x1', '0x3322']
```

NumPy - 副本和视图

在执行函数时，其中一些返回输入数组的副本，而另一些返回视图。 当内容物理存储在另一个位置时，称为副本。 另一方面，如果提供了相同内存内容的不同视图，我们将其称为视图。

无复制

简单的赋值不会创建数组对象的副本。 相反，它使用原始数组的相同 `id()` 来访问它。 `id()` 返回 Python 对象的通用标识符，类似于 C 中的指针。

此外，一个数组的任何变化都反映在另一个数组上。 例如，一个数组的形状改变也会改变另一个数组的形状。

示例

```
import numpy as np
a = np.arange(6)
print  '我们的数组是：'
print a
print  '调用 id() 函数：'
print id(a)
print  'a 赋值给 b：'
b = a
print b
print  'b 拥有相同 id()：'
print id(b)
print  '修改 b 的形状：'
b.shape =  3,2
print b
print  'a 的形状也修改了：'
print a
```

输出如下：

```
我们的数组是：
[0 1 2 3 4 5]

调用 id() 函数：
139747815479536

a 赋值给 b：
[0 1 2 3 4 5]
b 拥有相同 id()：
139747815479536

修改 b 的形状：
[[0 1]
 [2 3]
 [4 5]]

a 的形状也修改了：
[[0 1]
 [2 3]
 [4 5]]
```

视图或浅复制

NumPy 拥有 `ndarray.view()` 方法，它是一个新的数组对象，并可查看原始数组的相同数据。与前一种情况不同，新数组的维数更改不会更改原始数据的维数。

示例

```
import numpy as np
# 最开始 a 是个 3X2 的数组
a = np.arange(6).reshape(3,2)
print  '数组 a：'
print a
print  '创建 a 的视图：'
b = a.view()
print b
print  '两个数组的 id() 不同：'
print  'a 的 id()：'
print id(a)
print  'b 的 id()：'
print id(b)
# 修改 b 的形状，并不会修改 a
b.shape =  2,3
print  'b 的形状：'
print b
print  'a 的形状：'
print a
```

输出如下：

```
数组 a：
[[0 1]
 [2 3]
 [4 5]]

创建 a 的视图：
[[0 1]
 [2 3]
 [4 5]]

两个数组的 id() 不同：
a 的 id()：
140424307227264
b 的 id()：
140424151696288

b 的形状：
[[0 1 2]
 [3 4 5]]

a 的形状：
[[0 1]
 [2 3]
 [4 5]]
```

数组的切片也会创建视图：

示例

```
import numpy as np
a = np.array([[10,10],  [2,3],  [4,5]])
print  '我们的数组：'
print a
print  '创建切片：'
s = a[:,  :2]
print s
```

输出如下：

```
我们的数组：
[[10 10]
 [ 2 3]
 [ 4 5]]

创建切片：
[[10 10]
 [ 2 3]
 [ 4 5]]
```

深复制

`ndarray.copy()` 函数创建一个深层副本。 它是数组及其数据的完整副本，不与原始数组共享。

示例

```
import numpy as np
a = np.array([[10,10],  [2,3],  [4,5]])
print  '数组 a：'
print a
print  '创建 a 的深层副本：'
b = a.copy()
print  '数组 b：'
print b
# b 与 a 不共享任何内容
print  '我们能够写入 b 来写入 a 吗？'
print b is a
print  '修改 b 的内容：'
b[0,0]  =  100
print  '修改后的数组 b：'
print b
print  'a 保持不变：'
print a
```

输出如下：

```
数组 a：
[[10 10]
 [ 2 3]
 [ 4 5]]

创建 a 的深层副本：
数组 b：
[[10 10]
 [ 2 3]
 [ 4 5]]
我们能够写入 b 来写入 a 吗？
False

修改 b 的内容：
修改后的数组 b：
[[100 10]
 [ 2 3]
 [ 4 5]]

a 保持不变：
[[10 10]
 [ 2 3]
 [ 4 5]]
```

NumPy - 矩阵库

NumPy 包包含一个 Matrix库 `numpy.matlib` 。此模块的函数返回矩阵而不是返回 `ndarray` 对象。

`matlib.empty()`

`matlib.empty()` 函数返回一个新的矩阵，而不初始化元素。 该函数接受以下参数。

```
numpy.matlib.empty(shape, dtype, order)
```

其中：

序号	参数及描述
1.	`shape` 定义新矩阵形状的整数或整数元组
2.	`Dtype` 可选，输出的数据类型
3.	`order` `C` 或者 `F`

示例

```
import numpy.matlib
import numpy as np
print np.matlib.empty((2,2))
# 填充为随机数据
```

输出如下：

```
[[ 2.12199579e-314,   4.24399158e-314]
 [ 4.24399158e-314,   2.12199579e-314]]
```

`numpy.matlib.zeros()`

此函数返回以零填充的矩阵。

```
import numpy.matlib
import numpy as np
print np.matlib.zeros((2,2))
```

输出如下：

```
[[ 0.  0.]
 [ 0.  0.]])
```

numpy.matlib.ones()

此函数返回以一填充的矩阵。

```
import numpy.matlib
import numpy as np
print np.matlib.ones((2,2))
```

输出如下：

```
[[ 1.  1.]
 [ 1.  1.]]
```

numpy.matlib.eye()

这个函数返回一个矩阵，对角线元素为 1，其他位置为零。 该函数接受以下参数。

```
numpy.matlib.eye(n, M,k, dtype)
```

其中：

序号	参数及描述
1.	`n` 返回矩阵的行数
2.	`M` 返回矩阵的列数，默认为 `n`
3.	`k` 对角线的索引
4.	`dtype` 输出的数据类型

示例

```
import numpy.matlib
import numpy as np
print np.matlib.eye(n =  3, M =  4, k =  0, dtype =  float)
```

输出如下：

```
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]])
```

`numpy.matlib.identity()`

`numpy.matlib.identity()` 函数返回给定大小的单位矩阵。单位矩阵是主对角线元素都为 1 的方阵。

```
import numpy.matlib
import numpy as np
print np.matlib.identity(5, dtype =  float)
```

输出如下：

```
[[ 1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.]]
```

`numpy.matlib.rand()`

·numpy.matlib.rand()`函数返回给定大小的填充随机值的矩阵。

示例

```
import numpy.matlib
import numpy as np
print np.matlib.rand(3,3)
```

输出如下：

```
[[ 0.82674464  0.57206837  0.15497519]
 [ 0.33857374  0.35742401  0.90895076]
 [ 0.03968467  0.13962089  0.39665201]]
```

注意，矩阵总是二维的，而`ndarray`是一个 n 维数组。 两个对象都是可互换的。

示例

```
import numpy.matlib
import numpy as np

i = np.matrix('1,2;3,4')
print i
```

输出如下：

```
[[1  2]
 [3  4]]
```

示例

```
import numpy.matlib
import numpy as np

j = np.asarray(i)
print j
```

输出如下：

```
[[1  2]
 [3  4]]
```

示例

```
import numpy.matlib
import numpy as np

k = np.asmatrix (j)
print k
```

输出如下：

```
[[1  2]
 [3  4]]
```

NumPy - 线性代数

NumPy 包包含`numpy.linalg`模块，提供线性代数所需的所有功能。 此模块中的一些重要功能如下表所述。

序号	函数及描述
1.	`dot` 两个数组的点积
2.	`vdot` 两个向量的点积
3.	`inner` 两个数组的内积
4.	`matmul` 两个数组的矩阵积
5.	`determinant` 数组的行列式
6.	`solve` 求解线性矩阵方程
7.	`inv` 寻找矩阵的乘法逆矩阵

`numpy.dot()`

此函数返回两个数组的点积。 对于二维向量，其等效于矩阵乘法。 对于一维数组，它是向量的内积。 对于 N 维数组，它是`a`的最后一个轴上的和与`b`的倒数第二个轴的乘积。

```
import numpy.matlib
import numpy as np

a = np.array([[1,2],[3,4]])
b = np.array([[11,12],[13,14]])
np.dot(a,b)
```

输出如下：

```
[[37  40]
 [85  92]]
```

要注意点积计算为：

```
[[1*11+2*13, 1*12+2*14],[3*11+4*13, 3*12+4*14]]
```

numpy.vdot()

此函数返回两个向量的点积。 如果第一个参数是复数，那么它的共轭复数会用于计算。 如果参数 `id` 是多维数组，它会被展开。

例子

```
import numpy as np
a = np.array([[1,2],[3,4]])
b = np.array([[11,12],[13,14]])
print np.vdot(a,b)
```

输出如下：

```
130
```

注意： `1*11 + 2*12 + 3*13 + 4*14 = 130` 。

numpy.inner()

此函数返回一维数组的向量内积。 对于更高的维度，它返回最后一个轴上的和的乘积。

例子

```
import numpy as np
print np.inner(np.array([1,2,3]),np.array([0,1,0]))
# 等价于 1*0+2*1+3*0
```

输出如下：

```
2
```

例子

```
# 多维数组示例
import numpy as np
a = np.array([[1,2], [3,4]])

print '数组 a:'
print a
b = np.array([[11, 12], [13, 14]])

print '数组 b:'
print b

print '内积:'
print np.inner(a,b)
```

输出如下：

```
数组 a:
[[1 2]
[3 4]]

数组 b:
[[11 12]
[13 14]]

内积:
[[35 41]
[81 95]]
```

上面的例子中，内积计算如下：

```
1*11+2*12, 1*13+2*14
3*11+4*12, 3*13+4*14
```

numpy.matmul

`numpy.matmul()` 函数返回两个数组的矩阵乘积。 虽然它返回二维数组的正常乘积，但如果任一参数的维数大于2，则将其视为存在于最后两个索引的矩阵的栈，并进行相应广播。

另一方面，如果任一参数是一维数组，则通过在其维度上附加 1 来将其提升为矩阵，并在乘法之后被去除。

例子

```
# 对于二维数组，它就是矩阵乘法
import numpy.matlib
import numpy as np

a = [[1,0],[0,1]]
b = [[4,1],[2,2]]
print np.matmul(a,b)
```

输出如下：

```
[[4  1]
 [2  2]]
```

例子

```
# 二维和一维运算
import numpy.matlib
import numpy as np

a = [[1,0],[0,1]]
b = [1,2]
print np.matmul(a,b)
print np.matmul(b,a)
```

输出如下：

```
[1  2]
[1  2]
```

例子

```
# 维度大于二的数组
import numpy.matlib
import numpy as np

a = np.arange(8).reshape(2,2,2)
b = np.arange(4).reshape(2,2)
print np.matmul(a,b)
```

输出如下：

```
[[[2   3]
   [6   11]]
  [[10  19]
   [14  27]]]
```

`numpy.linalg.det()`

行列式在线性代数中是非常有用的值。 它从方阵的对角元素计算。 对于 2×2 矩阵，它是左上和右下元素的乘积与其他两个的乘积的差。

换句话说，对于矩阵`[[a，b]，[c，d]]`，行列式计算为`ad-bc`。 较大的方阵被认为是 2×2 矩阵的组合。

`numpy.linalg.det()`函数计算输入矩阵的行列式。

例子

```
import numpy as np
a = np.array([[1,2], [3,4]])
print np.linalg.det(a)
```

输出如下：

```
-2.0
```

例子

```
b = np.array([[6,1,1], [4, -2, 5], [2,8,7]])
print b
print np.linalg.det(b)
print 6*(-2*7 - 5*8) - 1*(4*7 - 5*2) + 1*(4*8 - -2*2)
```

输出如下：

```
[[ 6 1 1]
 [ 4 -2 5]
 [ 2 8 7]]

-306.0

-306
```

`numpy.linalg.solve()`

`numpy.linalg.solve()` 函数给出了矩阵形式的线性方程的解。

考虑以下线性方程：

```
x + y + z = 6

2y + 5z = -4

2x + 5y - z = 27
```

可以使用矩阵表示为：

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 5 \\ 2 & 5 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 6 \\ -4 \\ 27 \end{bmatrix}$$

如果矩阵成为 `A`、`X` 和 `B`，方程变为：

```
AX = B
```

或

```
X = A^(-1)B
```

`numpy.linalg.inv()`

我们使用 `numpy.linalg.inv()` 函数来计算矩阵的逆。矩阵的逆是这样的，如果它乘以原始矩阵，则得到单位矩阵。

例子

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.linalg.inv(x)
print x
print y
print np.dot(x,y)
```

输出如下：

```
[[1 2]

 [3 4]]

[[-2.   1. ]

 [ 1.5 -0.5]]

[[  1.00000000e+00   1.11022302e-16]

 [  0.00000000e+00   1.00000000e+00]]
```

例子

现在让我们在示例中创建一个矩阵A的逆。

```
import numpy as np
a = np.array([[1,1,1],[0,2,5],[2,5,-1]])

print '数组 a：'
print a
ainv = np.linalg.inv(a)

print 'a 的逆：'
print ainv

print '矩阵 b：'
b = np.array([[6],[-4],[27]])
print b

print '计算：A^(-1)B：'
x = np.linalg.solve(a,b)
print x
# 这就是线性方向 x = 5, y = 3, z = -2 的解
```

输出如下：

```
数组 a：
[[ 1 1 1]
 [ 0 2 5]
 [ 2 5 -1]]

a 的逆：
[[ 1.28571429 -0.28571429 -0.14285714]
 [-0.47619048 0.14285714 0.23809524]
 [ 0.19047619 0.14285714 -0.0952381 ]]

矩阵 b：
[[ 6]
 [-4]
 [27]]

计算：A^(-1)B：
[[ 5.]
 [ 3.]
 [-2.]]
```

结果也可以使用下列函数获取

```
x = np.dot(ainv,b)
```

NumPy - Matplotlib

Matplotlib 是 Python 的绘图库。 它可与 NumPy 一起使用，提供了一种有效的 MatLab 开源替代方案。 它也可以和图形工具包一起使用，如 PyQt 和 wxPython。

Matplotlib 模块最初是由 John D. Hunter 编写的。 自 2012 年以来，Michael Droettboom 是主要开发者。 目前，Matplotlib 1.5.1 是可用的稳定版本。 该软件包可以二进制分发，其源代码形式在 www.matplotlib.org 上提供。

通常，通过添加以下语句将包导入到 Python 脚本中：

```
from matplotlib import pyplot as plt
```

这里 `pyplot()` 是 matplotlib 库中最重要的函数，用于绘制 2D 数据。 以下脚本绘制方程 `y = 2x + 5` ：

示例

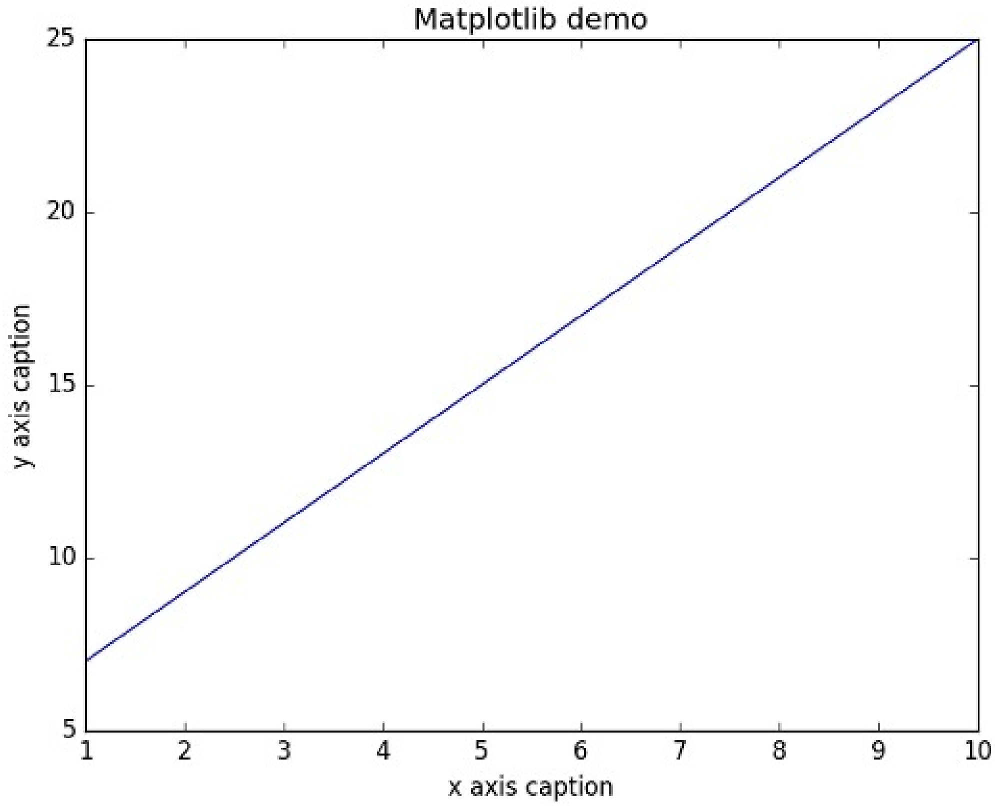
```
import numpy as np
from matplotlib import pyplot as plt

x = np.arange(1,11)
y =  2  * x +  5
plt.title("Matplotlib demo")
plt.xlabel("x axis caption")
plt.ylabel("y axis caption")
plt.plot(x,y) plt.show()
```

`ndarray` 对象 `x` 由 `np.arange()` 函数创建为 `x` 轴上的值。 `y` 轴上的对应值存储在另一个数组对象 `y` 中。 这些值使用 `matplotlib` 软件包的 `pyplot` 子模块的 `plot()` 函数绘制。

图形由 `show()` 函数展示。

上面的代码应该产生以下输出：



作为线性图的替代，可以通过向 `plot()` 函数添加格式字符串来显示离散值。 可以使用以下格式化字符。

字符	描述
'-'	实线样式
'--'	短横线样式
'-.'	点划线样式
':'	虚线样式
'.'	点标记
','	像素标记
'o'	圆标记
'v'	倒三角标记
'^'	正三角标记
'<'	左三角标记
'>'	右三角标记
'1'	下箭头标记
'2'	上箭头标记
'3'	左箭头标记
'4'	右箭头标记
's'	正方形标记
'p'	五边形标记
'*'	星形标记
'h'	六边形标记 1
'H'	六边形标记 2
'+'	加号标记
'x'	X 标记
'D'	菱形标记
'd'	窄菱形标记
'|'	竖直线标记
'_'	水平线标记

还定义了以下颜色缩写。

字符	颜色
`'b'`	蓝色
`'g'`	绿色
`'r'`	红色
`'c'`	青色
`'m'`	品红色
`'y'`	黄色
`'k'`	黑色
`'w'`	白色

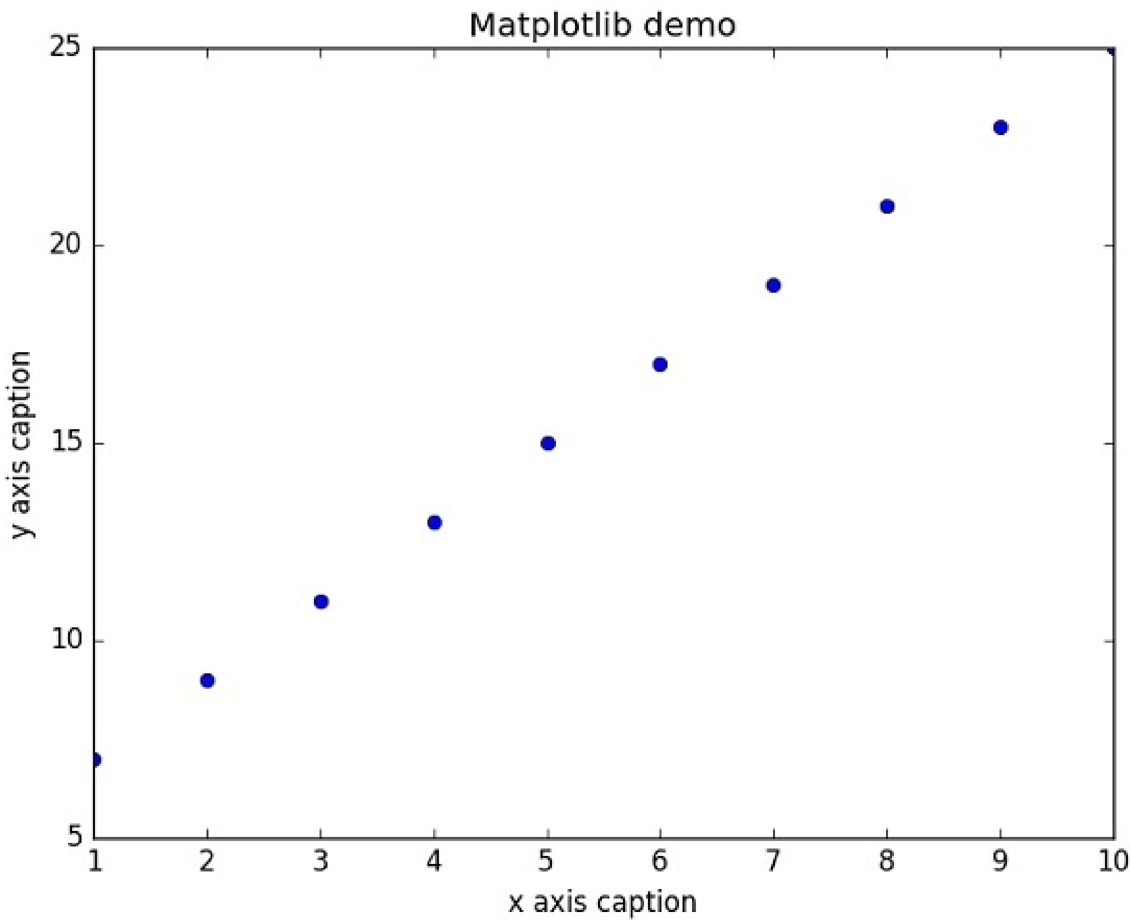
要显示圆来代表点，而不是上面示例中的线，请使用 `ob` 作为 `plot()` 函数中的格式字符串。

示例

```
import numpy as np
from matplotlib import pyplot as plt

x = np.arange(1,11)
y =  2  * x +  5
plt.title("Matplotlib demo")
plt.xlabel("x axis caption")
plt.ylabel("y axis caption")
plt.plot(x,y,"ob")
plt.show()
```

上面的代码应该产生以下输出：

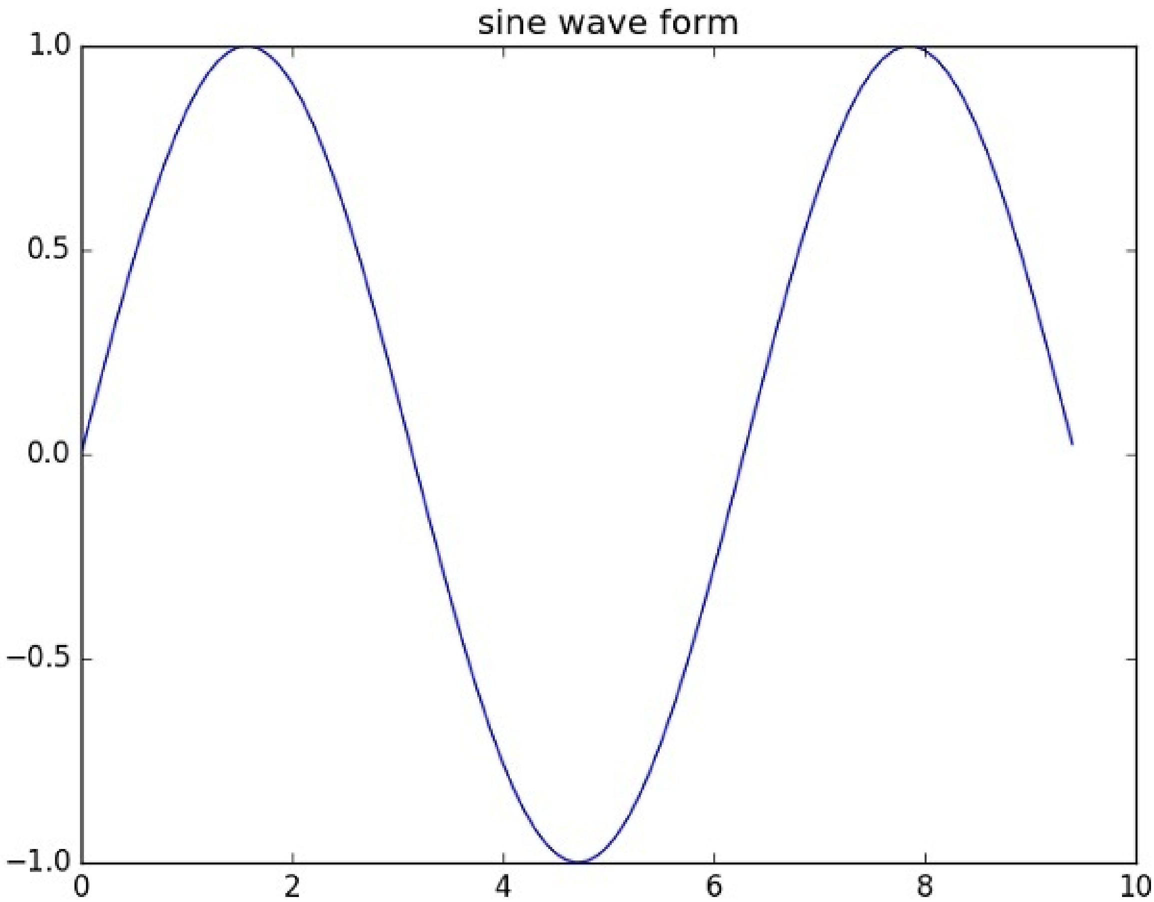


绘制正弦波

以下脚本使用 matplotlib 生成正弦波图。

示例

```
import numpy as np
import matplotlib.pyplot as plt
# 计算正弦曲线上点的 x 和 y 坐标
x = np.arange(0,  3  * np.pi,  0.1)
y = np.sin(x)
plt.title("sine wave form")
# 使用 matplotlib 来绘制点
plt.plot(x, y)
plt.show()
```



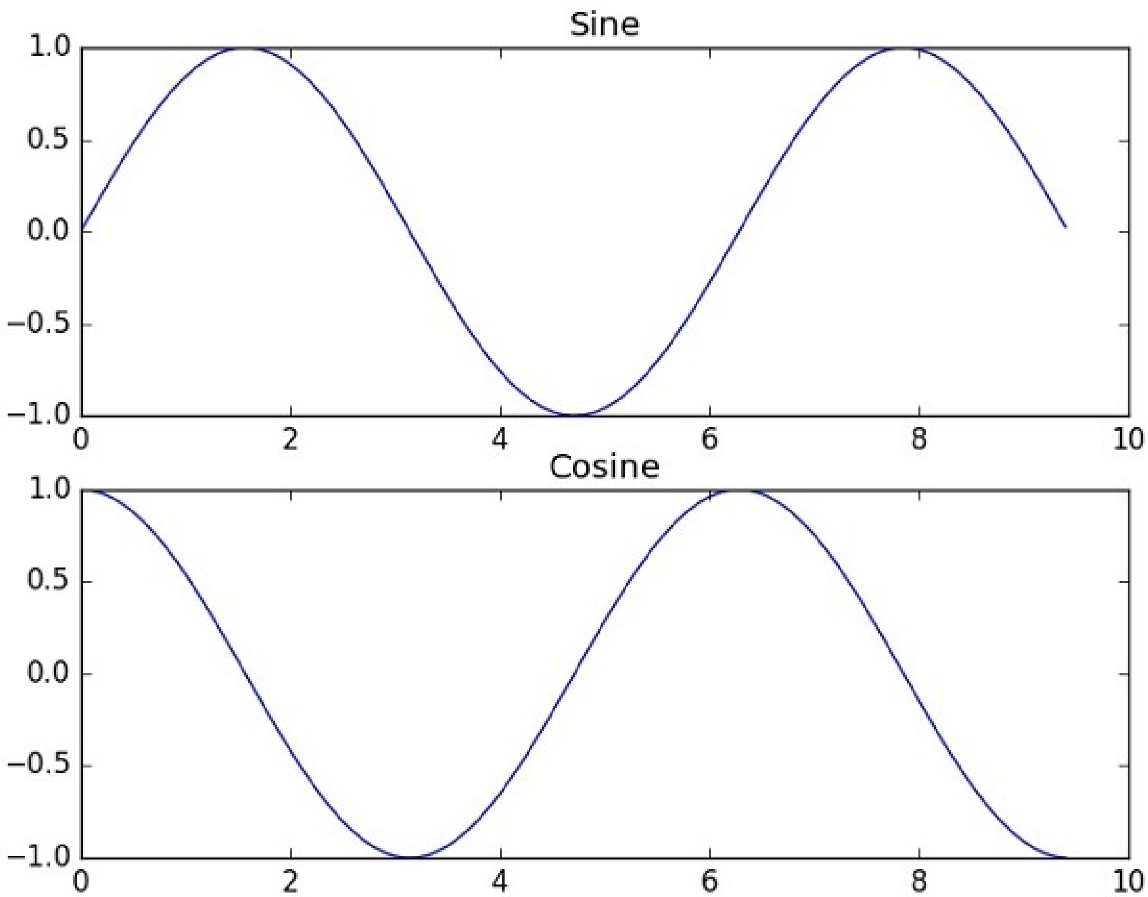
`subplot()`

`subplot()` 函数允许你在同一图中绘制不同的东西。 在下面的脚本中，绘制正弦和余弦值。

示例

```
import numpy as np
import matplotlib.pyplot as plt
# 计算正弦和余弦曲线上的点的 x 和 y 坐标
x = np.arange(0,  3  * np.pi,  0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)
# 建立 subplot 网格，高为 2，宽为 1
# 激活第一个 subplot
plt.subplot(2,  1,  1)
# 绘制第一个图像
plt.plot(x, y_sin)
plt.title('Sine')
# 将第二个 subplot 激活，并绘制第二个图像
plt.subplot(2,  1,  2)
plt.plot(x, y_cos)
plt.title('Cosine')
# 展示图像
plt.show()
```

上面的代码应该产生以下输出：



bar()

`pyplot` 子模块提供 `bar()` 函数来生成条形图。 以下示例生成两组 `x` 和 `y` 数组的条形图。

示例

```
from matplotlib import pyplot as plt
x =  [5,8,10]
y =  [12,16,6]
x2 =  [6,9,11]
y2 =  [6,15,7]
plt.bar(x, y, align =  'center')
plt.bar(x2, y2, color =  'g', align =  'center')
plt.title('Bar graph')
plt.ylabel('Y axis')
plt.xlabel('X axis')
plt.show()
```

NumPy - 使用 Matplotlib 绘制直方图

NumPy 有一个 `numpy.histogram()` 函数，它是数据的频率分布的图形表示。 水平尺寸相等的矩形对应于类间隔，称为 `bin` ，变量 `height` 对应于频率。

`numpy.histogram()`

`numpy.histogram()` 函数将输入数组和 `bin` 作为两个参数。 `bin` 数组中的连续元素用作每个 `bin` 的边界。

```
import numpy as np

a = np.array([22,87,5,43,56,73,55,54,11,20,51,5,79,31,27]) ]
np.histogram(a,bins =  [0,20,40,60,80,100])
hist,bins = np.histogram(a,bins =  [0,20,40,60,80,100])
print hist
print bins
```

输出如下：

```
[3 4 5 2 1]
[0 20 40 60 80 100]
```

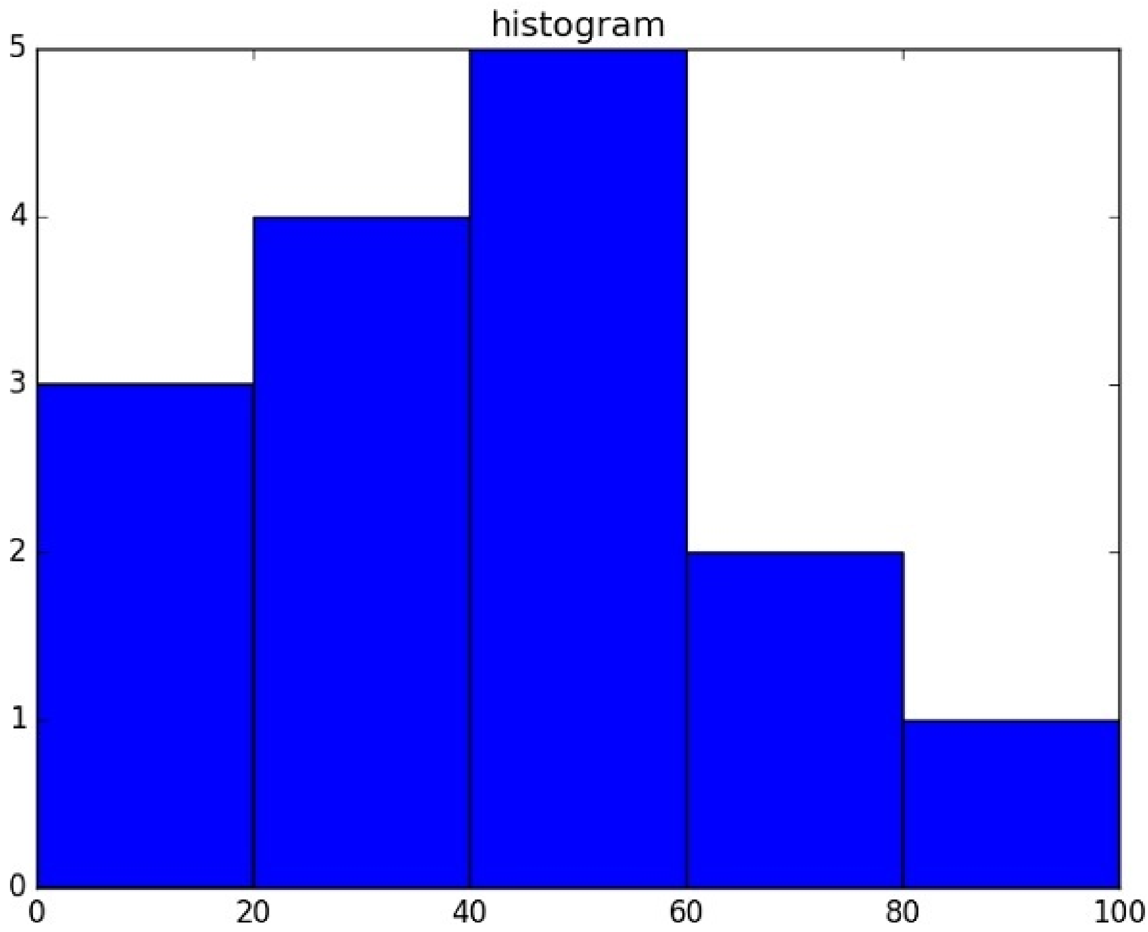
`plt()`

Matplotlib 可以将直方图的数字表示转换为图形。 `pyplot` 子模块的 `plt()` 函数将包含数据和 `bin` 数组的数组作为参数，并转换为直方图。

```
from matplotlib import pyplot as plt
import numpy as np

a = np.array([22,87,5,43,56,73,55,54,11,20,51,5,79,31,27])
plt.hist(a, bins =  [0,20,40,60,80,100])
plt.title("histogram")
plt.show()
```

输出如下：


histogram
5
4
3
2
1
0
0
20
40
60
80
100

NumPy - IO

`ndarray` 对象可以保存到磁盘文件并从磁盘文件加载。 可用的 IO 功能有：

- `load()` 和 `save()` 函数处理 numPy 二进制文件（带 `npy` 扩展名）
- `loadtxt()` 和 `savetxt()` 函数处理正常的文本文件

NumPy 为 `ndarray` 对象引入了一个简单的文件格式。 这个 `npy` 文件在磁盘文件中，存储重建 `ndarray` 所需的数据、图形、 `dtype` 和其他信息，以便正确获取数组，即使该文件在具有不同架构的另一台机器上。

`numpy.save()`

`numpy.save()` 文件将输入数组存储在具有 `npy` 扩展名的磁盘文件中。

```
import numpy as np
a = np.array([1,2,3,4,5])
np.save('outfile',a)
```

为了从 `outfile.npy` 重建数组，请使用 `load()` 函数。

```
import numpy as np
b = np.load('outfile.npy')
print b
```

输出如下：

```
array([1, 2, 3, 4, 5])
```

`save()` 和 `load()` 函数接受一个附加的布尔参数 `allow_pickles` 。 Python 中的 `pickle` 用于在保存到磁盘文件或从磁盘文件读取之前，对对象进行序列化和反序列化。

`savetxt()`

以简单文本文件格式存储和获取数组数据，是通过 `savetxt()` 和 `loadtxt()` 函数完成的。

示例

```
import numpy as np

a = np.array([1,2,3,4,5])
np.savetxt('out.txt',a)
b = np.loadtxt('out.txt')
print b
```

输出如下：

```
[ 1.  2.  3.  4.  5.]
```

`savetxt()` 和 `loadtxt()` 数接受附加的可选参数，例如页首，页尾和分隔符。

NumPy - 实用资源

以下资源包含有关 NumPy 的其他信息。 请使用它们获得更多的深入知识。

- 试验性 Numpy 教程
- 100 numpy exercises
- From Python to Numpy
- Matplotlib 教程